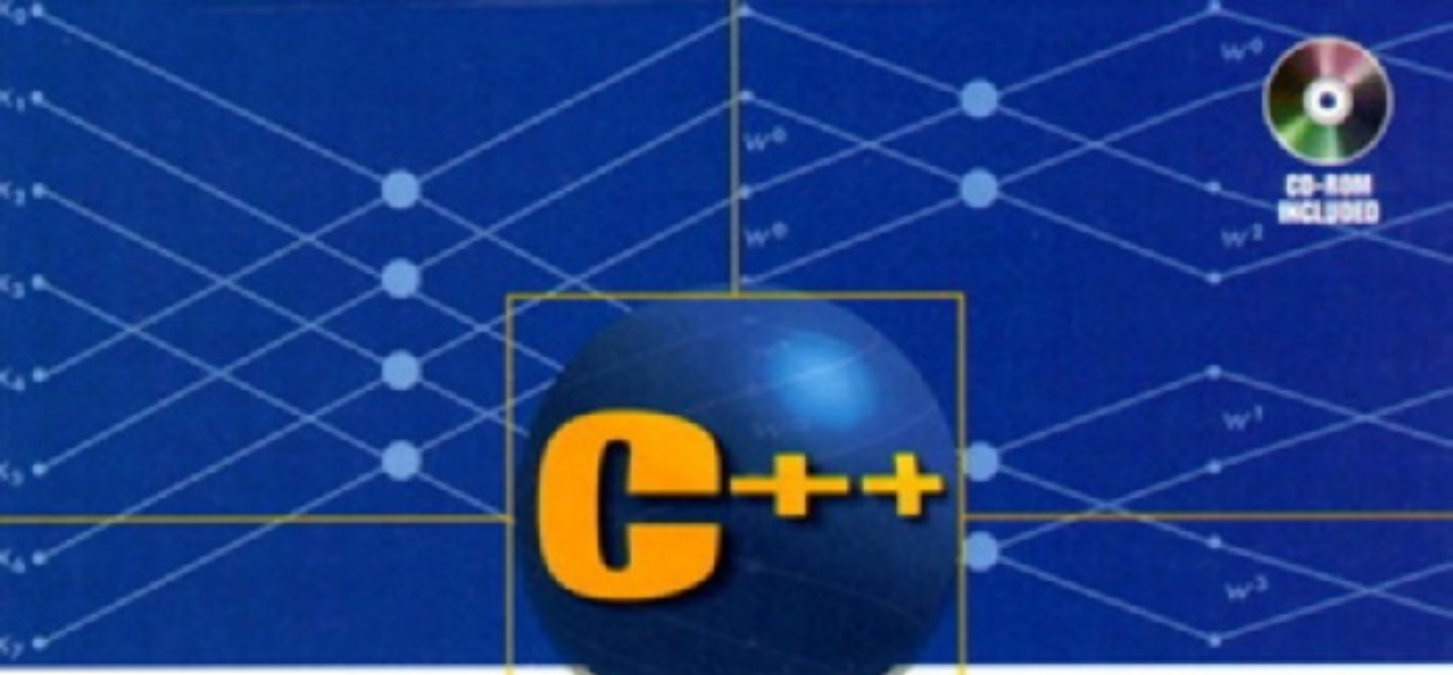


ALGORITHMS *for* DIGITAL SIGNAL PROCESSING

Second Edition

Paul M. Embree • Damon Danieli

C++ Algorithms for Digital Signal Processing

ISBN 0-13-179144-3

This page intentionally left blank

C++ Algorithms for Digital Signal Processing

Paul M. Embree

Damon Danieli

Prentice Hall PTR
Upper Saddle River, New Jersey 07458

Embree, Paul M.
C++ algorithms for digital signal processing / by Paul M. Embree, Damon Danieli.
p. cm.
Incudes bibliographical references and index.
ISBN 0-13-179144-3
1. Signal processing—Digital techniques. 2. C++ (Computer program language) 3. Computer algorithms. I. Danieli, Damon. II. Title.
TK5102.9 E45 1999
621.382′2′02855133—dc 21 98-28681
CIP

Editorial/Production Supervision: *Kerry Reardon*
Acquisitions Editor: *Bernard Goodwin*
Buyer: *Alan Fischer*
Cover Design: *Amy Rosen*
Cover Design Direction: *Jerry Votta*
Marketing Manager: *Kaylie Smith*

Prentice-Hall, Inc.
Upper Saddle River, NJ 07458

The publisher offers discounts on this book when ordered in bulk quantities. For more information, call the Corporate Sales Department at 800-382-3419; FAX: 201-236-714, email corpsales@prenhall.com or write Corporate Sales Department, Prentice Hall PTR, One Lake Street, Upper Saddle River, NJ 07458

Prentice Hall books are widely used by corporations and government agencies for training, marketing, and resale.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-179144-3

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

This page intentionally left blank

Preface

This book is written with the conviction that two current trends in engineering and programming will continue in the foreseeable future and will become very closely related. The first trend is the rapidly growing importance of digital signal processing (DSP). Digital techniques have become the method of choice in signal processing as digital computers have increased in power, speed, and convenience and as powerful microprocessors have become more available. Some examples of the applications of DSP to engineering problems are:

- Radar signal processing such as
  Synthetic aperture radar imaging
  Multitarget tracking
  Radar classification and identification
- Ultrasound and sonar signal processing such as
  Doppler flow measurement
  Adaptive beam forming
  Image display and enhancement
- Image processing such as
  Target recognition
  Pattern classification
  Robot vision
  Image compression and restoration

- Communications signal processing such as
  Frequency hopped signal tracking
  Spread spectrum signal recovery
  Signal modulation and demodulation
  Adaptive equalization
- Geological signal processing such as
  Formation identification
  Structure velocity estimation
- Speech signal processing such as
  Short-time spectral analysis
  Speaker independent word recognition
  Phoneme identification
  Speech synthesis

As DSP has engulfed signal processing, the C language is proving itself to be the most valuable programming tool for real-time and computationally intensive software tasks. Due to the nature of DSP, this second trend is related in very important ways to the first. There are two broad areas of software applications in DSP:

Applications where the software is used to simulate hardware
Applications where the software is an end product in itself

The C and C++ languages are reasonably high-level languages suitable for either of these areas. They have aspects of high-level languages that make them suitable for simulation work and still allow the engineer to produce code whose efficiency approaches that of assembly language for real-time applications.

The C and C++ languages have significant advantages for DSP applications over other languages such as FORTRAN and Pascal. One important reason is the utility of C data structures and C++ objects for signal processing tasks. Also, the inherent modularity of C and C++ is a valuable asset in DSP programming. Digital signal processing repetitively uses a well-defined set of mathematical tools with small parameter variations. The ordering and tailoring of these algorithms to specific applications are the art of DSP. The C and C++ languages are constructed to encourage development of external library routines and objects that can be used as building blocks in the exact way required by DSP.

Another reason the C++ language is a good choice for DSP is the popularity and widespread use of this language. Compilers are available for all popular microprocessors including 32-bit designs. In addition, many manufacturers of digital signal processing devices (such as Texas Instruments, AT&T, Motorola, and Analog Devices) provide C compilers for both 16-bit integer and 32-bit floating-point signal processing integrated circuits. The code produced by the best compilers is compact and efficient, and there are sufficient common features among compilers to allow portable code to be written if the standard ANSI C conventions are used. This allows the C code for DSP algorithms to be used directly in embedded real-time signal processing systems. All of the programs in

this book are suitable for use with any standard ANSI C compiler on UNIX systems, IBM-PC platforms, and many real-time programming environments.

Although C++ has not been the traditional language of real-time embedded systems programmers, it has been growing in popularity for application development and fast prototyping of designs. Not only does C++ allow the programmer to fully encapsulate the data with the methods that operate on the data, its inherent modularity makes it easy to write good code. Just one look at the C++ implementation of complex math operations in Chapter 5 or vectors and matrices in Chapter 6 should give the reader some idea about the power and flexibility the language offers.

This book is constructed in such a way that it will be most useful to the professional engineer, student, and hobbyist who is familiar with both digital signal processing and programming but who is not necessarily an expert in both. This book is intended to be the ideal tool to help the reader in developing efficient, compact, and accurate programs for use in a particular DSP application. In addition, any reader who has the need to write DSP programs will be assisted by the combination of theory and actual working programs presented here. The book is useful for students of DSP and fast numerical techniques because of the numerous examples of efficient DSP algorithms and numerous exercises at the end of each chapter. The book can also serve as a quick source of debugged programs for the applications-oriented programmer who wishes to supplement an existing C or C++ library. For readers interested in a complete DSP software library, the programs presented in the text are available in a machine-readable form on the CD-ROM disk included with the book. C language versions of the C++ programs discussed in the text are also included on the CD-ROM for users of microprocessors that do not have C++ compilers availible.

The text is divided into several sections. Chapters 1 and 2 cover the basic principles to digital signal processing and C++ programming. Readers familiar with these topics may wish to skip one or both chapters. Chapter 3 covers basic use of the DSP programs, the data file formats, and user interface that will be used throughout the text. Chapters 4 and 5 cover basic one-dimensional digital signal processing techniques. Digital filtering is presented in Chapter 4, and frequency domain techniques are discussed in Chapter 5. Chapter 6 describes a C++ implementation of vectors and matrices that can be used for one-dimensional or two-dimensional signal processing. Chapter 7 discusses two-dimensional signal processing using algorithms described in Chapters 1.

The CD-ROM disk included with the text contains source code for all of the DSP programs and DSP data associated with the examples discussed in this book. Appendix B and the file `README.TXT` on the disk provide more information about how to compile and run the programs. The programs are also precompiled and stored on the CD-ROM to allow the reader to try the un-modified examples on an IBM-PC platform without installing a C++ compiler. These programs have been tested using Microsoft® Visual C++® Compiler version 4.2, 5.0 and 6.0 (Microsoft® Visual C++® verson 6.0, Introductory Edition is included on the CD-ROM), but will also work with most modern compilers that adhere to the ANSI C/C++ standards.

Paul M. Embree
Damon Danieli

This page intentionally left blank

List of Key Symbols

A Φ	Matrices are denoted by uppercase bold Latin and greek letters
AR	Autoregressive
ARMA	Autoregressive-moving average
B	Equivalent bandwidth in hertz
$c_{xy}[m]$	Cross covariance of discrete processes $x[n]$ and $y[n]$ at lag m
cov $\{xy\}$	Covariance between random variables x and y
d γ	Vectors are denoted by lowercase bold Latin and Greek letters
det **A**	Determinant of matrix **A**
DFT	Discrete time Fourier transform
E$\{x[n]\}$	Expectation operation on random process $x[n\}$
$\mathcal{F}\{x(t)\} = X(f)$	Continuous Fourier transform of continuous time signal $x(t)$
$\mathcal{F}^{-1}\{X(f)\} = x(t)$	Inverse continuous Fourier transform of $X(f)$
f	Frequency in hertz
f_s	Frequency sampling rate in hertz
FFT	Fast Fourier transform
I	Identity matrix
Im$\{x\}$	Imaginary part of complex variable x
j	Imaginary constant, $\sqrt{-1}$
J	Reflection (exchange) matrix
MA	Moving average
MLE	Maximum likelihood estimate
mse$\{x\}$	Mean square error of random variable x

n	Discrete time or space index
N	Number of data samples
$p(x)$	Probability density function of random variable x
$P_{xx}(f)$	Power spectral density
$P_{xy}(f)$	Cross-power spectral density
PDF	Probability density function
PSD	Power spectral density
$r_{xx}[m]$	Autocorrelation sequence at lag m
$r_{xy}[m]$	Cross-correlation sequence at lag m
$\mathbf{R}$	Autocorrelation matrix composed of $r_{xx}[0]$ to $r_{xx}[p]$
$\mathrm{Re}\{x\}$	Real part of complex variable x
s	Continuous space variable (units of ft, m, km, etc.)
$S_{xx}(f)$	Energy spectral density
t	Continuous-time variable (units of seconds)
T	Temporal sampling interval in seconds
$\mathbf{T}$	Toeplitz matrix
$\mathrm{var}\{x\}$	Variance of random variable x
$x[n]$	Discrete time (or space) function
$x(t)$	Continuous-time (or space) function
WSS	Wide-sense stationary
z	Complex variable
$\mathcal{Z}\{h[n]\} = \mathrm{H}(z)$	z-transform of $h[n]$
$\mathcal{Z}^{-1}\{\mathrm{H}(z)\} = h[n]$	Inverse z-transform of $\mathrm{H}(z)$
$\mathbf{0}$	Vector of zero elements
$\delta(t)$	Continuous delta function
$u_0(n)$	Discrete delta function
^{T}	Vector of matrix transposition
^{H}	Vector and matrix complex conjugate transposition
*	Scalar, vector, and matrix complex conjugation
$\star$	Convolution operator

C++ Algorithms for Digital Signal Processing

This page intentionally left blank

CHAPTER 1

Digital Signal Processing Fundamentals

Digital signal processing begins with a discrete-time quantized signal that appears to the computer as a sequence of digital values. Figure 1.1 shows an example of a digital signal processing operation. There is an input sequence $x(n)$, the operator $\mathbb{O}\{\ \}$, and an output sequence, $y(n)$. Operators may be classified as *linear* or *nonlinear* and Figure 1.2 shows where the operators described in this book fit in such a classification (linearity will be discussed in Section 1.2).

Operators are applied to sequences in order to effect the following results:

1. Extract parameters or features from the sequence.
2. Produce a similar sequence with particular features enhanced or eliminated.
3. Restore the sequence to some earlier state.
4. Encode or compress the sequence.

A complete digital signal processing system may consist of many operations on the same sequence as well as operations on the result of operations. An example is shown in Figure 1.3.

This chapter is divided into several sections. Section 1.1 deals with *sequences* of numbers: where and how they originate, their spectra, and their relation to continuous signals. Section 1.2 describes the common characteristics of *linear time invariant operators,* which are the most often used in DSP. Section 1.3 discusses the class of operators called *digital filters*. Section 1.4 introduces the *discrete Fourier transform* (DFT). Section 1.5 describes the properties of commonly used *nonlinear operators*. Sections 1.6 and 1.7 cover basic *linear algebra* and *probability theory* and apply these to signal processing.

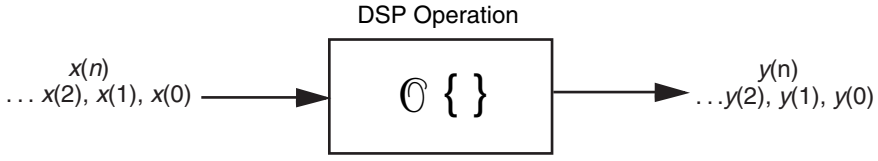


FIGURE 1.1 DSP operator.

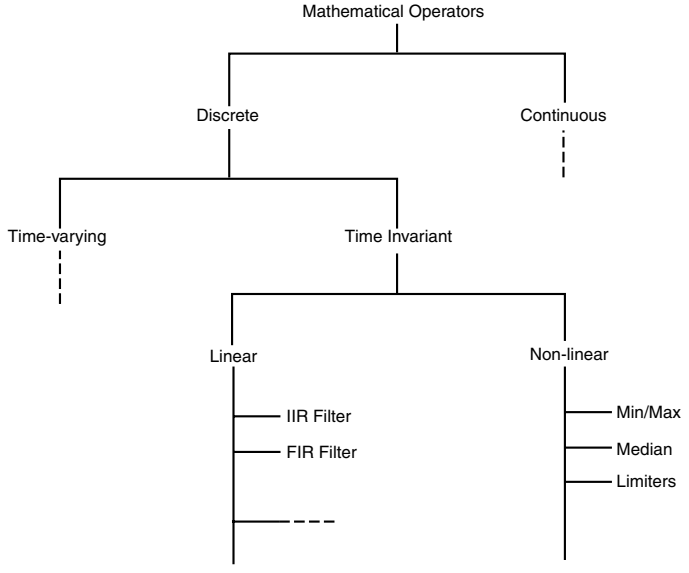


FIGURE 1.2 Tree structure of operators.

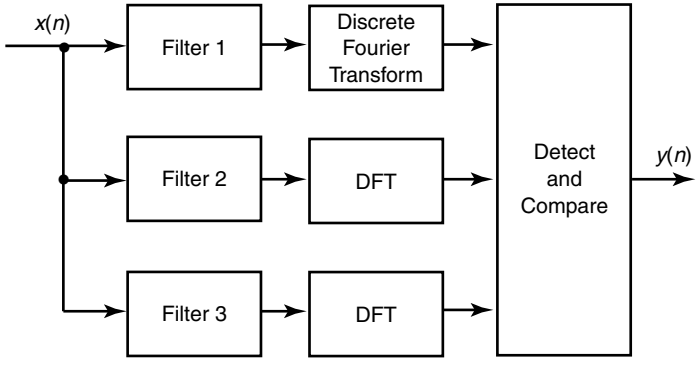


FIGURE 1.3 DSP system.

Section 1.8 introducts *adaptive filters* and systems. Finally, Section 1.9 extends the material on *one-dimensional signal processing* into *two-dimensional image processing.*

1.1 SEQUENCES

In order for the digital computer to manipulate a signal, the signal must have been sampled at some interval. Figure 1.4 shows an example of a continuous function of time that has been sampled at intervals of T seconds. The resulting set of numbers is called a *sequence.* If the continuous-time function was $x(t)$, then the samples would be $x(nT)$ for $n,$ an integer extending over some finite range of values. The independent variable, $t,$ could just as well have been a spatial coordinate, say, $x,$ making the continuous variable function $f(x)$ and the sampled function $f(nX)$, where X is the sampling interval in distance. Extending to two dimensions, the continuous function could be $f(x,y)$, possibly a two-dimensional image representation of a scene. Two-dimensional signal processing will be discussed in Section 1.9.

It is common practice to normalize the sample interval to 1 and drop it from the equations. The sequence then becomes $x(n)$. Care must be taken, however, when calculating power or energy from the sequences. The sample interval, including units of time or space, must be reinserted at the appropriate points in the power or energy calculations.

A sequence as a representation of a continuous-time signal has the following important characteristics:

1. The signal is sampled. It has finite value at only discrete points in time.
2. The signal is truncated outside some finite length representing a finite time interval.
3. The signal is quantized. It is limited to discrete steps in amplitude, where the step size and, therefore, the accuracy (or *signal fidelity*) depend on the arithmetic precision of the computer.

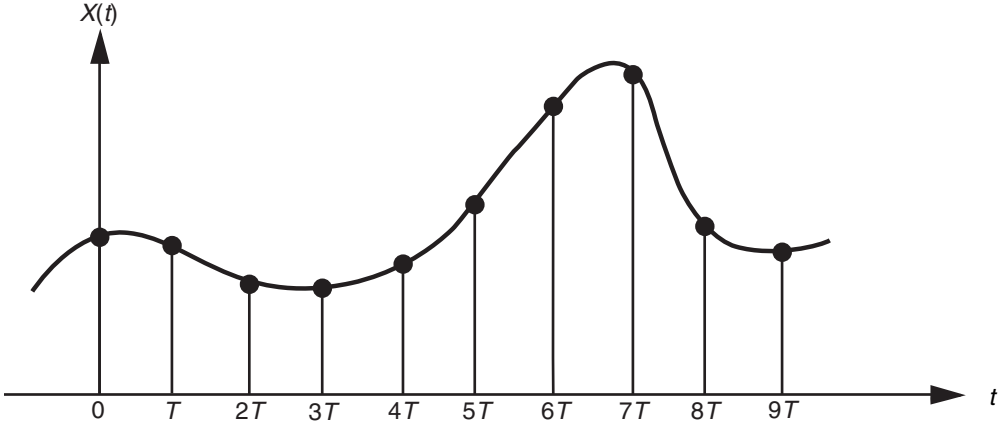


FIGURE 1.4 Sampling.

In order to understand the nature of the results that DSP operators produce, these characteristics must be taken into account. The effect of sampling will be considered in Section 1.1.1. Truncation will be considered in the section on the discrete Fourier transform (Section 1.4), and quantization will be discussed in Section 1.7.4.

1.1.1 The Sampling Function

The *sampling function* is the key to traveling between the continuous-time and discrete time worlds. It is called by various names: the *Dirac delta function,* the *sifting function,* the *singularity function,* and the *sampling function* among them. It has the following properties:

$$\text{Property 1.} \int_{-\infty}^{\infty} f(t)\delta(t-\tau)dt = f(\tau) \tag{1.1}$$

$$\text{Property 2.} \int_{-\infty}^{\infty} \delta(t-\tau)dt = 1 \tag{1.2}$$

In the equations above, τ can be any real number.

To see how this function can be thought of as the ideal sampling function, first consider the realizable sampling function, $\Delta(t)$, illustrated in Figure 1.5. Its pulse width is one unit of time, and its amplitude is one unit of amplitude. It clearly exhibits Property 2 of the sampling function. When $\Delta(t)$ is multiplied by the function to be sampled, however, the $\Delta(t)$ sampling function chooses not a single instant in time but a range from –½ to +½. As a result, Property 1 of the sampling function is not met. Instead, the following integral would result:

$$\int_{-\infty}^{\infty} f(t)\Delta(t-\tau)dt = \int_{\tau-\frac{1}{2}}^{\tau+\frac{1}{2}} f(t)dt. \tag{1.3}$$

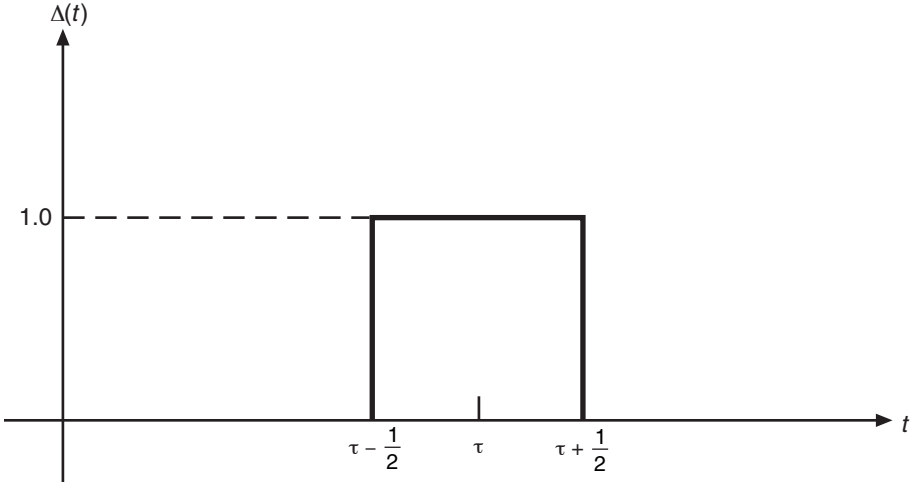


FIGURE 1.5 Realizable sampling function.

This can be thought of as a kind of smearing of the sampling process across a band that is related to the pulse width of $\Delta(t)$. A better approximation to the sampling function would be a function $\Delta(t)$ with a narrower pulse width. As the pulse width is narrowed, however, the amplitude must be increased. In the limit, we can see that the ideal sampling function must have infinitely narrow pulse width so that it samples at a single instant in time, and infinitely large amplitude so that the sampled signal still contains the same finite energy.

Figure 1.4 illustrates the sampling process using the sampling function at sample intervals of T. The resulting time waveform can be written

$$x_s(t) = \sum_{n=-\infty}^{\infty} x(t)\delta(t - nT). \tag{1.4}$$

The waveform that results from this process is impossible to visualize due to the infinite amplitude and zero width of the ideal sampling function. It may be easier for the reader to picture a somewhat less than ideal sampling function (one with very small width and very large amplitude) multiplying the continuous time waveform.

It should be emphasized that $x_s(t)$ is a continuous time waveform made from the superposition of an infinite set of continuous time signals $x\,(t)\,\delta\,(t - nT)$. It can also be written

$$x_s(t) = \sum_{n=-\infty}^{\infty} x(nT)\delta(t - nT) \tag{1.5}$$

since the sampling function gives a nonzero multiplier only at the values $t = nT$. In this last equation, the sequence $x(nt)$ makes its appearance. This is the set of numbers we actually would like to deal with, and the next sections will show how this is possible.

1.1.2 Sampled Signal Spectra

Using *Fourier transform theory,* the frequency spectrum of the continuous time waveform $x(t)$ can be written

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt \tag{1.6}$$

and the time waveform can be expressed in terms of its spectrum as

$$x(t) = \int_{-\infty}^{\infty} X(f)e^{j2\pi ft}df \tag{1.7}$$

Since this is true for any continuous function of time, $x(t)$, it is also true for $x_s(t)$.

$$X_s(f) = \int_{-\infty}^{\infty} x_s(t)e^{-j2\pi ft}dt \tag{1.8}$$

Replacing $x_s(t)$ by the sampling representation

$$X_s(f) = \int_{-\infty}^{\infty} \left[\sum_{n=-\infty}^{\infty} x(t)\delta(t - nT) \right] e^{-j2\pi ft} dt \tag{1.9}$$

The order of the summation and integration can be interchanged and Property 1 of the sampling function applied to give

$$X_s(f) = \sum_{n=-\infty}^{\infty} x(nT) e^{-j2\pi fnT} \tag{1.10}$$

This equation is the exact form of a Fourier series representation of $X_s(f)$, a periodic function of frequency having period $1/T$. The coefficients of the Fourier series are $x(nT)$, and they can be calculated from the following integral:

$$x(nT) = T \int_{\frac{1}{-2T}}^{\frac{1}{2T}} X_s(f) e^{j2\pi fnT} df \tag{1.11}$$

The last two equations are a Fourier series pair that allow calculation of either the time signal or frequency spectrum in terms of the opposite member of the pair. Notice that the use of the problematic signal $x_s(t)$ is eliminated, and the sequence $x(nT)$ can be used instead.

1.1.3 Continuous- and Discrete Time Signal Spectra

By evaluating Equation 1.7 at $t = nT$ and setting the result equal to the right-hand side of Equation 1.11 the following relationship between the two spectra is obtained:

$$x(nT) = \int_{-\infty}^{\infty} X(f) e^{j2\pi fnT} df = T \int_{\frac{1}{-2T}}^{\frac{1}{2T}} X_s(f) e^{j2\pi fnT} df \tag{1.12}$$

The right-hand side of Equation 1.7 can be expressed as the infinite sum of a set of integrals with finite limits

$$x(nT) = \sum_{m=-\infty}^{\infty} T \int_{\frac{2m-1}{2T}}^{\frac{2m+1}{2T}} X(f) e^{j2\pi fnT} df \tag{1.13}$$

By changing variables to $\lambda = f - m/T$ (substituting $f = \lambda + m \neq T$)

$$x(nT) = \sum_{m=-\infty}^{\infty} \int_{-\frac{1}{2T}}^{\frac{1}{2T}} X(\lambda + \frac{m}{T}) e^{j2\pi\lambda nT} e^{j2\pi \frac{m}{T} nT} d\lambda \tag{1.14}$$

Moving the summation inside the integral, recognizing that $e^{j2\pi mn}$ (for all integers m and n) is equal to 1, and equating everything inside the integral to the similar part of Equation 1.11 gives the following relation:

$$X_s(f) = \sum_{m=-\infty}^{\infty} X(f + \frac{m}{T}) \tag{1.15}$$

Equation 1.15 shows that the sampled time frequency spectrum is equal to an infinite sum of shifted replicas of the continuous time frequency spectrum overlaid on each other. The shift of the replicas is equal to the sample frequency, $1/T$. It is interesting to examine the conditions under which the two spectra are equal to each other, at least for a limited range of frequencies. In the case where there are no spectral components of frequency greater than $f = \frac{1}{2T}$ in the original continuous time waveform, the two spectra are equal over the frequency range $f = -\frac{1}{2T}$ to $f = \frac{1}{2T}$. Of course, the sampled time spectrum will repeat this same set of amplitudes periodically for all frequencies, while the continuous time spectrum is identically zero for all frequencies outside the specified range.

The *Nyquist sampling criterion* is based on the derivation just presented and asserts that a continuous time waveform, when sampled at a frequency greater than twice the maximum frequency component in its spectrum, can be reconstructed completely from the sampled waveform. Conversely, if a continuous time waveform is sampled at a

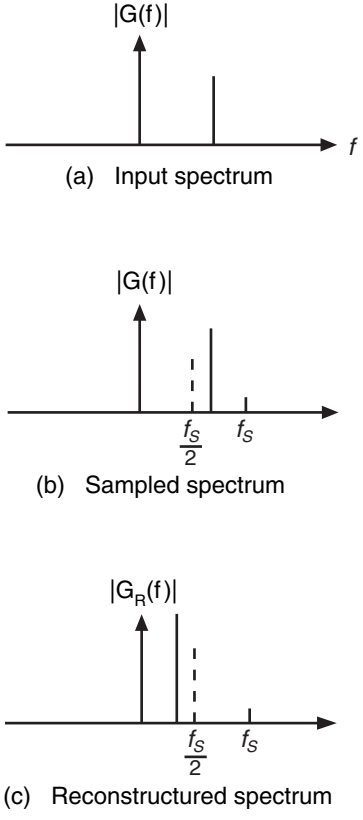


FIGURE 1.6 Aliasing in the frequency domain. (a) Input spectrum. (b) Sampled spectrum. (c) Reconstructed spectrum.

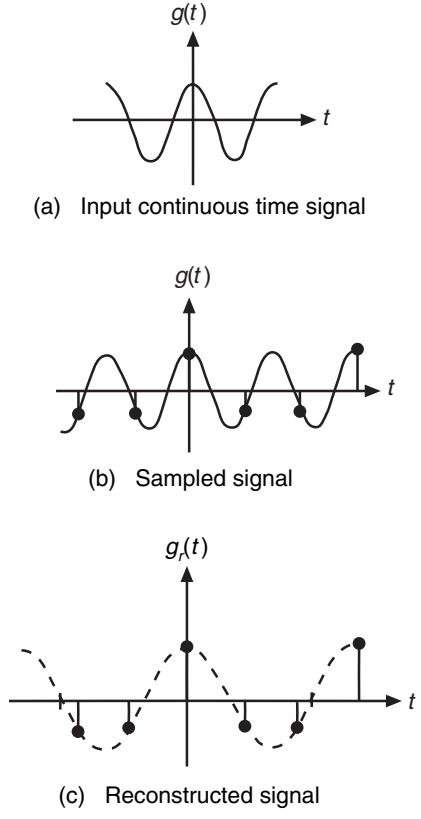


FIGURE 1.7 Aliasing in the time domain. (a) Input continuous time signal. (b) Sampled signal (c) Reconstructed signal.

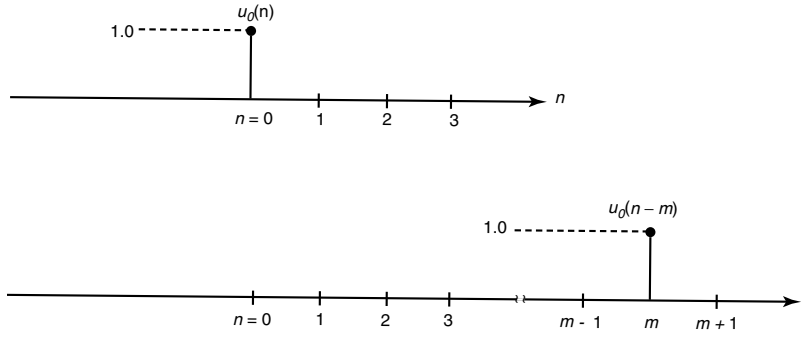


FIGURE 1.8 Shifting of the impulse sequence.

frequency lower than twice its maximum frequency component a phenomenon called *aliasing* occurs. If a continuous time signal is reconstructed from an aliased representation, distortions will be introduced into the result and the degree of distortion is dependent on the degree of aliasing. Figure 1.6(a) through 1.6(c) shows the spectra of sampled signals without aliasing and with aliasing. Figure 1.7(a) through 1.7(c) shows the reconstructed waveforms of an aliased signal.

1.1.4 The Impulse Sequence

There is a unique and important sequence called the *impulse sequence* and is represented by $u_0(n)$. The impulse sequence consists of an infinite number of zero-valued samples at all values of n except $n = 0$. At $n = 0$ the impulse sequence has a sample of amplitude 1. Any sequence can be shifted by subtracting a constant from its index, and so Figure 1.8 shows the impulse sequence and a shifted version: $u_0(n–m)$. The impulse sequence is the discrete time counterpart of the sampling function.

The properties of the impulse sequence can be listed in a similar fashion to the properties of the sampling function as follows:

$$\text{Property 1.} \sum_{n-\infty}^{\infty} x(n)u_0(n-m) = x(m) \tag{1.16}$$

$$\text{Property 2.} \sum_{n=-\infty}^{\infty} u_0(n-m) = 1 \tag{1.17}$$

The impulse function and its unique properties will be used in the next section on linear time invariant operators.

1.2 LINEAR TIME INVARIANT OPERATORS

The most commonly used DSP operators are *linear* and *time invariant* (or LTI). The linearity property is stated as follows:

Given $x(n)$, a finite sequence, and $\mathbb{O}\{\ \}$, an operator in n-space, let

$$y(n) = \mathbb{O}\{x(n)\} \tag{1.18}$$

If

$$x(n) = ax_1(n) + bx_2(n) \tag{1.19}$$

where a and b are constant with respect to n, then, if $\mathbb{O}\{\ \}$ is a linear operator

$$y(n) = a\mathbb{O}\{x_1(n)\} + b\mathbb{O}\{x_2(n)\} \tag{1.20}$$

The time invariant property means that if

$$y(n) = \mathbb{O}\{x(n)\}$$

then the shifted version gives the same response or

$$y(n-m) = \mathbb{O}\{x(n-m)\} \tag{1.21}$$

Another way to state this property is that if $x(n)$ is periodic with period N such that

$$x(n+N) = x(n)$$

then if $\mathbb{O}\{\ \}$ is a time invariant operator in n space

$$\mathbb{O}\{x(n+N)\} = \mathbb{O}\{x(n)\}$$

Next the LTI properties of the operator $\mathbb{O}\{\ \}$ will be used to derive an expression and method of calculation for $\mathbb{O}\{x(n)\}$. First the impulse sequence can be used to represent $x(n)$ in a different manner:

$$x(n) = \sum_{m=-\infty}^{\infty} x(m)u_0(n-m) \tag{1.22}$$

This is because

$$u_0(n-m) = \begin{cases} 1, & n = m \\ 0, & \text{otherwise} \end{cases} \tag{1.23}$$

The impulse sequence acts as a sampling or sifting function on the function $x(m)$, using the dummy variable m to sift through and find the single desired value $x(n)$. Now this somewhat devious representation of $x(n)$ is substituted into the operator Equation 1.18:

$$y(n) = \mathbb{O}\left\{\sum_{m=-\infty}^{\infty} x(m)u_0(n-m)\right\} \tag{1.24}$$

Recalling that $\mathbb{O}\{\ \}$ operates only on functions of n and using the linearity property

$$y(n) = \sum_{m=-\infty}^{\infty} x(m)\mathbb{O}\{u_0(n-m)\} \tag{1.25}$$

Every operator has a set of outputs that are its response when an impulse sequence is applied to its input. We will represent this impulse response by $h(n)$ so that

$$h(n) = \mathbb{O}\{u_0(n)\} \tag{1.26}$$

This impulse response is a sequence that has special significance for $\mathbb{O}\{\ \}$, since it is the sequence that occurs at the output of the block labeled $\mathbb{O}\{\ \}$ in Figure 1.1 when an impulse sequence is applied at the input. By time invariance it must be true that

$$h(n-m) = \mathbb{O}\{u_0(n-m)\} \tag{1.27}$$

so that

$$y(n) = \sum_{m=-\infty}^{\infty} x(m)\, h(n-m) \tag{1.28}$$

Equation 1.28 states that $y(n)$ is equal to the convolution of $x(n)$ with the impulse response $h(n)$. By substituting $m = n - p$ into Equation 1.28 an equivalent form is derived:

$$y(n) = \sum_{p=-\infty}^{\infty} h(p)\, x(n-p) \tag{1.29}$$

It must be remembered that m and p are dummy variables and are used for purposes of the summation only. From the equations just derived it is clear that the impulse response completely characterizes the operator $\mathbb{O}\{\}$ and can be used to label the block representing the operator as in Figure 1.9.

1.2.1 Causality

In the mathematical descriptions of sequences and operators thus far, it was assumed that the impulse responses of operators may include values that occur before any applied input stimulus. This is the most general form of the equations and has been suitable for the development of the theory to this point. However, it is clear that no physical system can produce an output in response to an input that has not yet been applied. Since DSP operators and sequences have their basis in physical systems, it is more useful to limit our discussion to that subset of operators and sequences that can exist in the real world.

The first step in representing realizable sequences is to acknowledge that any sequence must have started at some time. Thus, it is assumed that any element of a sequence in a realizable system whose time index is less than zero has a value of zero. Sequences that start at times later than this can still be represented, since an arbitrary

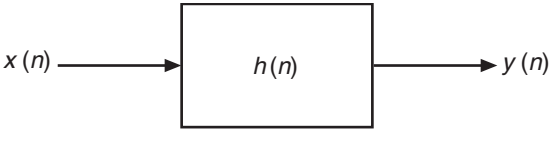


FIGURE 1.9 Impulse response representation of an operator.

number of their beginning values can also be zero. However, the earliest true value of any sequence must be at a value of n that is greater than or equal to zero. This attribute of sequences and operators is called *causality,* since it allows all attributes of the sequence to be caused by some physical phenomenon. Clearly, a sequence that has already existed for infinite time lacks a cause as the term is generally defined.

Thus, the convolution relation for causal operators becomes

$$y(n) = \sum_{m=0}^{\infty} h(m)\, x(n-m) \tag{1.30}$$

This form follows naturally, since the impulse response is a sequence and can have no values for m less than zero.

1.2.2 Difference Equations

All discrete time, linear, causal, time invariant operators can be described in theory by the Nth order difference equation

$$\sum_{m=0}^{N-1} a_m y(n-m) = \sum_{p=0}^{N-1} b_p x(n-p) \tag{1.31}$$

where $x(n)$ is the stimulus for the operator and $y(n)$ is the results or output of the operator. The equation remains completely general if all coefficients are normalized by the value of a_0 giving

$$y(n) + \sum_{m=1}^{N-1} a_m y(n-m) = \sum_{p=0}^{N-1} b_p x(n-p) \tag{1.32}$$

and the equivalent form

$$y(n) = \sum_{p=0}^{N-1} b_p x(n-p) - \sum_{m=1}^{N-1} a_m y(n-m) \tag{1.33}$$

or

$$\begin{aligned} y(n) = {} & b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) \ldots \\ & + b_{N-1} x(n-N+1) - a_1 y(n-1) - a_2 y(n-2) \\ & - \ldots - a_{N-1} y(n-N+1) \end{aligned} \tag{1.34}$$

To represent an operator properly may require a very high value of N, and for some complex operators N may have to be infinite. In practice, the value of N is kept within limits manageable by a computer, and there are often approximations made of a particular operator to make N an acceptable size. Also, it should be noted that the number of nonzero a_m coeffi-

cients may not be equal to the number of nonzero b_p coefficients. We will see examples in Section 1.3 where there may be fewer coefficients of x than of y and also vice versa.

In Equations 1.32 and 1.33 the terms $y(n - m)$ and $x(n - p)$ are shifted or delayed versions of the functions $y(n)$ and $x(n)$, respectively. For instance, Figure 1.10 shows a sequence $x(n)$ and $x(n - 3)$, which is the same sequence delayed by three sample periods. Using this delaying property and Equation 1.34, a structure or flow graph can be constructed for the general form of a discrete time LTI operator. This structure is shown in Figure 1.11. Each of the boxes is a delay element with unity gain. The coefficients are shown next to the legs of the flow graph to which they apply. The circles enclosing the summation symbol (Σ) are adder elements.

1.2.3 The *z*-transform Description of Linear Operators

There is a linear transform that is as useful to discrete time analysis as the Laplace transform is to continuous time analysis. This transform is called the *z-transform* and its definition is

$$\mathcal{Z}\{x(n)\} = \sum_{n=0}^{\infty} x(n)z^{-n} \tag{1.35}$$

where the symbol $\mathcal{Z}\{\ \}$ stands for "z-transform of" and the z in the equation is a complex number. One of the most important properties of the z-transform is its relationship to time delay in sequences. To show this property take a sequence, $x(n)$, with z-transform

$$\mathcal{Z}\{x(n)\} = X(z) = \sum_{n=0}^{\infty} x(n)z^{-n} \tag{1.36}$$

A shifted version of this sequence has a z-transform

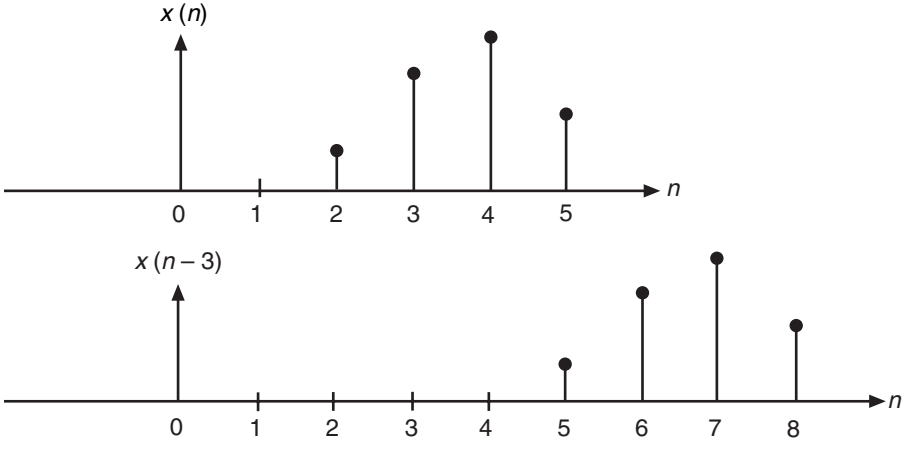


FIGURE 1.10 Shifting of a sequence.

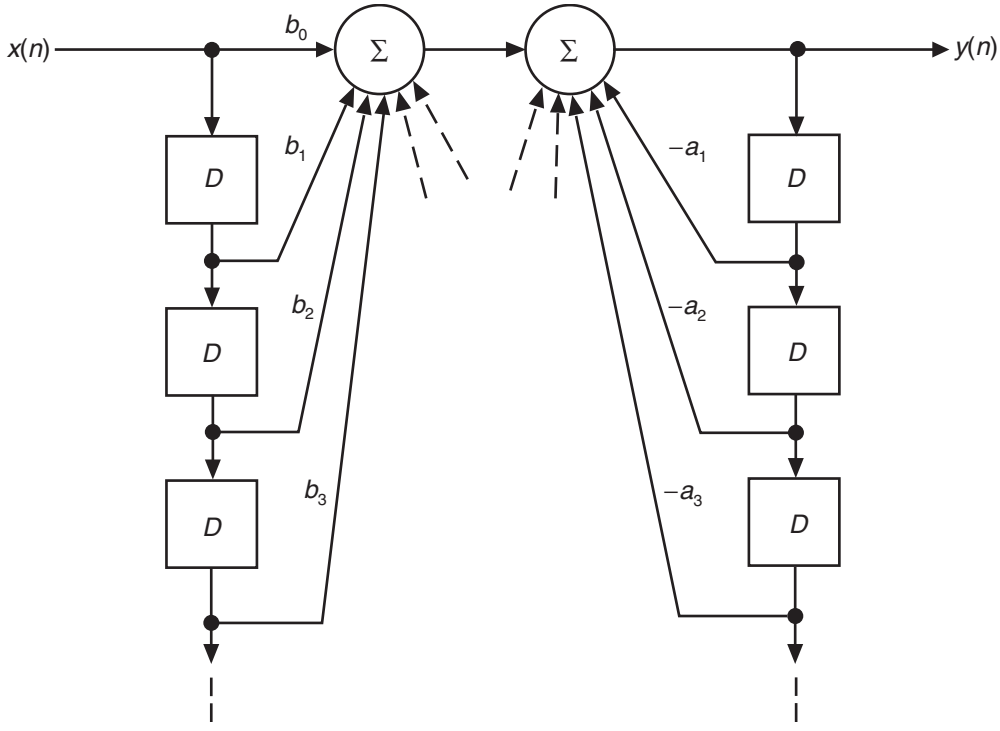


FIGURE 1.11 Flow graph structure of linear operators.

$$\mathcal{Z}\{x(n-p)\} = \sum_{n=0}^{\infty} x(n-p)z^{-n} \tag{1.37}$$

and since p is always taken to be a positive integer and $x(n)$ is 0 for $n < 0$

$$\mathcal{Z}\{x(n-p)\} = \sum_{n-p=0}^{\infty} x(n-p)z^{-n} \tag{1.38}$$

Now we make a variable substitution letting $m = n - p$ and therefore $n = m + p$. This gives:

$$\mathcal{Z}\{x(n-p)\} = \sum_{m=0}^{\infty} x(m)z^{-(m+p)} \tag{1.39}$$

$$= z^{-p} \sum_{m=0}^{\infty} x(m)z^{-m} \tag{1.40}$$

But comparing the summation in this last equation to Equation 1.35 for the z-transform of $x(n)$ it can be seen that

$$\mathcal{Z}\{x(n-p)\} = z^{-p}\mathcal{Z}\{x(n)\} = z^{-p}X(z) \tag{1.41}$$

We can apply this property of the z-transform to the general equation for LTI operators as follows:

$$\mathcal{Z}\left\{y(n) + \sum_{p=1}^{\infty} a_p y(n-p)\right\} = z^{-p}\ \mathcal{Z}\left\{\sum_{q=0}^{\infty} b_q x(n-q)\right\} \tag{1.42}$$

Since the z-transform is a linear transform, it possesses the distributive and associative properties and we can rewrite the equation:

$$\mathcal{Z}\{y(n)\} + \sum_{p=1}^{\infty} a_p \mathcal{Z}\{y(n-p)\} = \sum_{q=0}^{\infty} b_q \mathcal{Z}\{x(n-p)\} \tag{1.43}$$

Using the shift property of the z-transform (Equation 1.41):

$$Y(z) + \sum_{p=1}^{\infty} a_p z^{-p} Y(z) = \sum_{q=0}^{\infty} b_q z^{-q} X(z) \tag{1.44}$$

$$Y(z)\left[1 + \sum_{p=1}^{\infty} a_p z^{-p}\right] = X(z)\left[\sum_{q=0}^{\infty} b_q z^{-q}\right] \tag{1.45}$$

Finally, we can rearrange Equation 1.45 to give the transfer function in the z-transform domain:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{q=0}^{\infty} b_q z^{-q}}{1 + \sum_{p=1}^{\infty} a_p z^{-p}} \tag{1.46}$$

Using Equation 1.44, Figure 1.11 can be redrawn in the z-transform domain and this structure is shown in Figure 1.12. The flow graphs are identical if it is understood that a multiplication by z^{-1} in the transform domain is equivalent to a delay of one sampling time interval in the time domain.

1.2.4 Frequency Domain Transfer Function of an Operator

Taking the Fourier transform of both sides of Equation 1.30 (which describes any LTI causal operator) results in the following:

$$\mathcal{F}\{y(n)\} = \sum_{m=0}^{\infty} h(m)\mathcal{F}\{x(n-m)\} \tag{1.47}$$

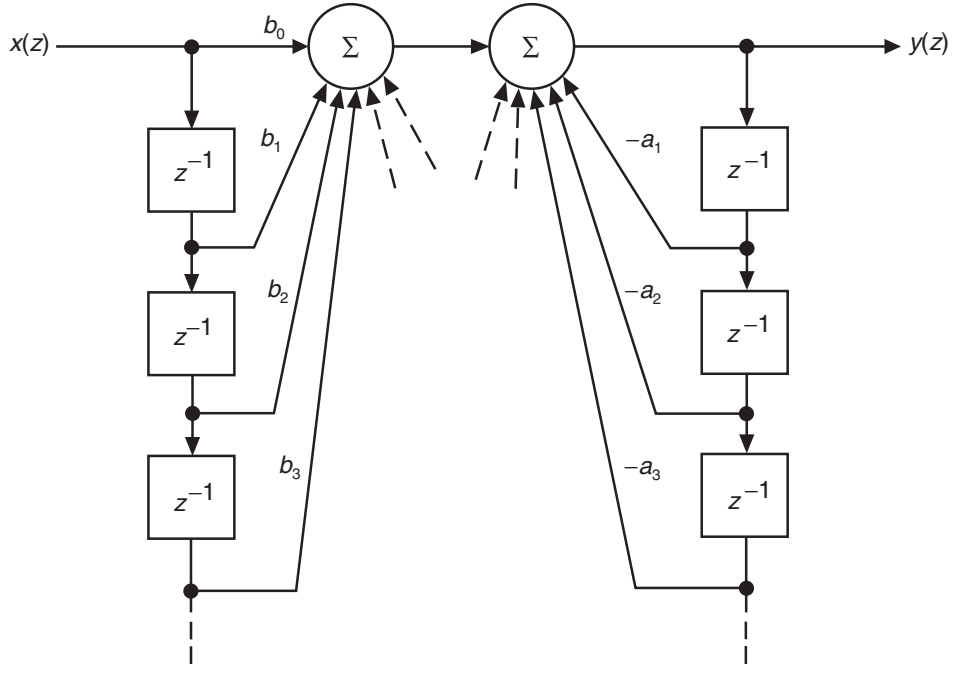


FIGURE 1.12 Flow graph structure for the *z*-transform of an operator.

Using one of the properties of the Fourier transform

$$\mathcal{F}\{x(n-m)\} = e^{-j2\pi fm}\mathcal{F}\{x(n)\} \tag{1.48}$$

From Equation 1.48 it follows that

$$Y(f) = \sum_{m=0}^{\infty} h(m)e^{-j2\pi fm}X(f) \tag{1.49}$$

or dividing both sides by $X(f)$

$$\frac{Y(f)}{X(f)} = \sum_{m=0}^{\infty} h(m)e^{-j2\pi fm} \tag{1.50}$$

which is easily recognized as the Fourier transform of the series $h(m)$. Rewriting this equation

$$\frac{Y(f)}{X(f)} = H(f) = \mathcal{F}\{h(m)\} \tag{1.51}$$

Figure 1.13 shows the time domain block diagram of Equation 1.51 and Figure 1.14 shows the Fourier transform (or frequency domain) block diagram and equation. The fre-

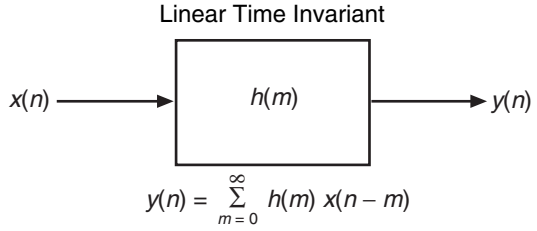


FIGURE 1.13 Time domain block diagram of LTI system.

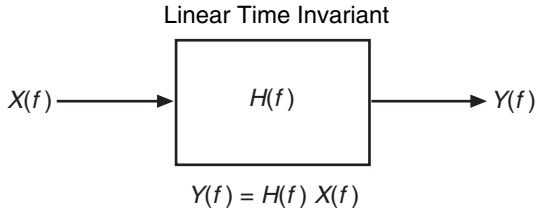


FIGURE 1.14 Frequency block diagram of LTI system.

quency domain description of a linear operator is often used to describe the operator. Most often it is shown as an amplitude and a phase angle plot as a function of the variable f (sometimes normalized with respect to the sampling rate, $1/T$).

1.2.5 Frequency Response Relationship to the *z*-transform

Recall the Fourier transform pair

$$X_s(f) = \sum_{n=-\infty}^{\infty} x(nT)e^{-j2\pi fnT} \tag{1.52}$$

and

$$x(nT) = \int_{-\infty}^{\infty} X_s(f)e^{j2\pi fnT}\,df \tag{1.53}$$

In order to simplify the notation we normalize the value of T, the period of the sampling waveform, to be equal to one.

Now compare Equation 1.52 to the equation for the z-transform of $x(n)$ as follows:

$$X(z) = \sum_{n=0}^{\infty} x(n)z^{-n} \tag{1.54}$$

Equations 1.52 and 1.54 are equal for sequences $x(n)$ that are causal [i.e., $x(n) = 0$ for all $n < 0$] if z is set as follows:

$$z = e^{j2\pi f} \tag{1.55}$$

A plot of the locus of values for z in the complex plane described by Equation 1.55 is shown in Figure 1.15. The plot is a circle of unit radius. Thus, the z-transform of a causal sequence, $x(n)$, when evaluated on the unit circle in the complex plane is equivalent to the

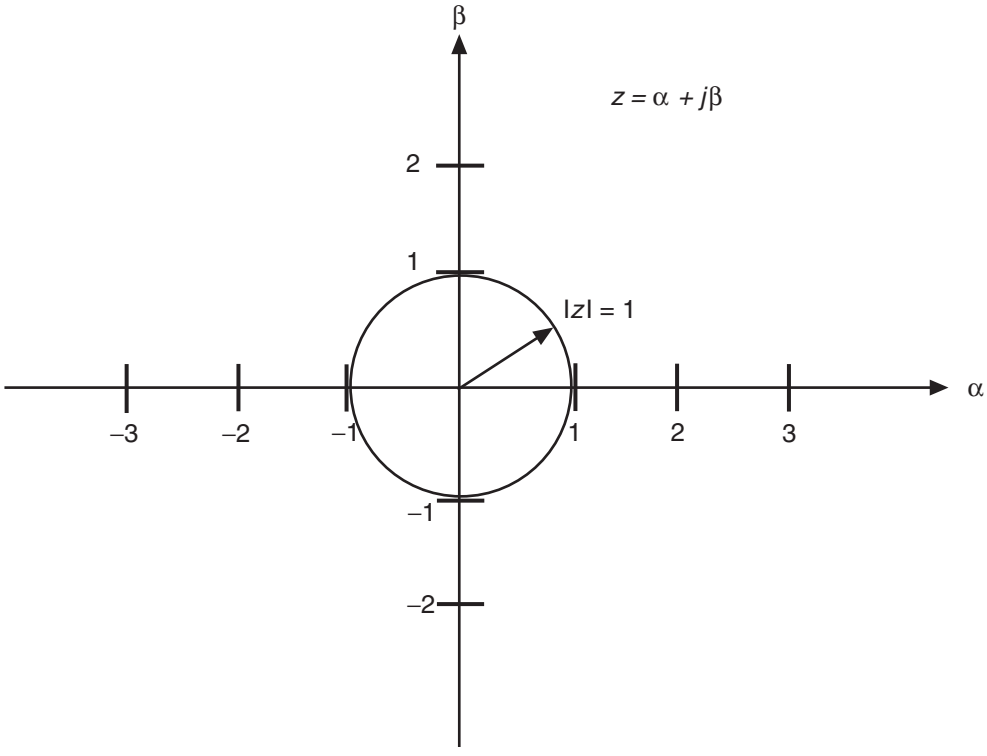


FIGURE 1.15 The unit circle in the *z*-plane.

frequency domain representation of the sequence. This is one of the properties of the *z*-transform that makes it very useful for discrete signal analysis.

Summarizing the last few paragraphs, the impulse response of an operator is simply a sequence, $h(m)$, and the Fourier transform of this sequence is the frequency response of the operator. The *z*-transform of the sequence $h(m)$, called $H(z)$, can be evaluated on the unit circle to yield the frequency domain representation of the sequence. This can be written as follows:

$$H(z)\Big|_{z=e^{j2\pi f}} = H(f) \tag{1.56}$$

1.2.6 Summary of Linear Operators

In Section 1.2 we have developed several representations for the most commonly used class of operators in discrete time systems: linear, causal, time invariant operators. The representations are summarized below:

1. Impulse response—$h(m)$ (Section 1.2.1). Associated equation:

$$y(n) = \sum_{m=0}^{\infty} h(m)x(n-m)$$

2. Frequency domain transfer function—$H(f)$ (Section 1.2.4). Associated equation:

$$Y(f) = H(f)X(f)$$

3. Difference equation representation (Section 1.2.2). Associated equation:

$$y(n) = \sum_{q=0}^{\infty} b_q x(n-q) - \sum_{p=1}^{\infty} a_p y(n-p)$$

4. z-transform transfer function—$H(z)$ (Section 1.2.3). Associated equation:

$$Y(z) = H(z)X(z)$$

Each of these representations is useful in the study of discrete time systems. An understanding of their relationships is one of the keys to efficient design of DSP systems.

1.3 DIGITAL FILTERS

The linear operators that have been presented and analyzed in the previous sections can be thought of as *digital filters.* The concept of *filtering* is an analogy between the action of a physical strainer or sifter to the action of a linear operator on sequences when the operator is viewed in the frequency domain. Such a filter might allow certain frequency components of the input to pass unchanged to the output while blocking other components. Naturally, any such action will have its corresponding result in the time domain. This view of linear operators opens a wide area of theoretical analysis and provides increased understanding of the action of digital systems.

There are two broad classes of digital filters. Recall the difference equation for a general operator:

$$y(n) = \sum_{q=0}^{Q-1} b_q x(n-q) - \sum_{\mathrm{p}=1}^{\mathrm{p}-1} \mathrm{a}_{\mathrm{p}} y(n-\mathrm{p}) \tag{1.57}$$

Notice that the infinite sums have been replaced with finite sums. This is necessary in order for the filters to be physically realizable.

The first class of digital filters have a_p equal to 0 for all p. The common name for filters of this type is *finite impulse response* (FIR) filters, since their response to an impulse dies away in a finite number of samples. These filters are also called *moving average* (or MA) filters, since the output is simply a weighted average of the input values.

$$y(n) = \sum_{q=0}^{Q-1} b_q x(n-q) \tag{1.58}$$

There is a window of these weights (b_q) that takes exactly the Q most recent values of $x(n)$ and combines them to produce the output.

The second class of digital filters are *infinite impulse response* (IIR) filters. This class includes both *autoregressive* (AR) filters and the most general form, *ARMA* filters. In the AR case, all b_q for $q = 1$ to $Q - 1$ are set to 0.

$$y(n) = x(n) - \sum_{p=1}^{P-1} a_p y(n-p) \tag{1.59}$$

For ARMA filters, the more general Equation 1.57 applies. In either type of IIR filter, a single-impulse response at the input can continue to provide output of infinite duration with a given set of coefficients. Stability can be a problem for IIR filters, since with poorly chosen coefficients the output can grow without bound for some inputs.

1.3.1 FIR Filters

Restating the general equation for FIR filters

$$y(n) = \sum_{q=0}^{Q-1} b_q x(n-q) \tag{1.60}$$

Comparing this equation with the convolution relation for linear operators

$$y(n) = \sum_{m=0}^{\infty} h(m)\, x(n-m)$$

one can see that the coefficients in an FIR filter are identical to the elements in the impulse response sequence if this impulse response is finite in length.

$$b_q = h(q) \quad \text{for } q = 0,1,2,3,\ldots,Q-1$$

This means that if one is given the impulse response sequence for a linear operator with a finite impulse response, one can immediately write down the FIR filter coefficients. However, as was mentioned at the start of this section, filter theory looks at linear operators primarily from the frequency domain point of view. Therefore, one is most often given the desired frequency domain response and asked to determine the FIR filter coefficients.

There are a number of methods for determining the coefficients for FIR filters given the frequency domain response. The two most popular FIR filter design methods are listed and described briefly below.

1. *Use of the DFT on the sampled frequency response.* In this method the required frequency response of the filter is sampled at a frequency interval of $1/T$, where T is the time between samples in the DSP system. The inverse discrete Fourier transform (see Section 1.4) is then applied to this sampled response to produce the impulse response of the filter. Best results are usually achieved if a smoothing window is applied to the frequency response before the inverse DFT is performed. A

simple method to obtain FIR filter coefficients based on the Kaiser window is described in Section 4.2.1 in Chapter 4.

2. *Optimal mini-max approximation using linear programming techniques.* There is a well-known program written by Parks and McClellan that uses the Remez exchange algorithm to produce an optimal set of FIR filter coefficients given the required frequency response of the filter. The Parks-McClellan program is available on the IEEE digital signal processing tape or as part of many of the filter design packages available for personal computers. The program is also printed in several DSP texts (see Elliot or Rabiner and Gold). The program REMEZ.C is a C language implementation of the Parks-McClellan program and is included on the enclosed disk. An example of a filter designed using the REMEZ program is shown in Section 4.2 in Chapter 4.

The design of digital filters will not be considered in detail here. Interested readers may wish to consult references listed at the end of this chapter that give complete descriptions of all the popular techniques.

The frequency response of FIR filters can be investigated by using the transfer function developed for a general linear operator:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{q=0}^{Q-1} b_q z^{-q}}{1 + \sum_{p=1}^{P-1} a_p z^{-p}} \tag{1.61}$$

Notice that the sums have been made finite to make the filter realizable. Since for FIR filters the a_p are all equal to 0, the equation becomes

$$H(z) = \frac{Y(z)}{X(z)} = \sum_{q=0}^{Q-1} b_q z^{-q} \tag{1.62}$$

The Fourier transform or frequency response of the transfer function is obtained by letting $z = {}^{j2\pi f}$, which gives

$$H(f) = H(z)\Big|_{z=e^{j2\pi f}} = \sum_{q=0}^{Q-1} b_q e^{-j2\pi fq} \tag{1.63}$$

This is a polynomial in powers of z^{-1} or a sum of products of the form

$$H(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3} + \ldots + b_{Q-1} z^{-(Q-1)}$$

There is an important class of FIR filters for which this polynomial can be factored into a product of sums from

$$H(z) = \prod_{m=0}^{M-1} (z^{-2} + \alpha_m z^{-1} + \beta_m) \prod_{n=0}^{N-1} (z^{-1} + \gamma_n) \tag{1.64}$$

This expression for the transfer function makes explicit the values of the variable z^{-1} that cause $H(z)$ to become zero. These points are simply the roots of the quadratic equations

$$0 = z^{-2} + \alpha_m z^{-1} + \beta_m$$

which in general provide complex conjugate zero pairs, and the values γ_n, which provide single zeroes.

1.3.2 Linear Phase in FIR Filters

In many communication and image processing applications it is essential to have filters whose transfer functions exhibit a phase characteristic that changes linearly with a change in frequency. This characteristic is important because it is the phase transfer relationship that gives minimum distortion to a signal passing through the filter. A very useful feature of FIR filters is that for a simple relationship of the coefficients, b_q, the resulting filter is guaranteed to have linear phase response. The derivation of the relationship that provides a linear phase filter follows.

A linear phase relationship to frequency means that

$$H(f) = |H(f)|\, e^{j[\alpha f + \beta]}$$

where α and ß are constants. If the transfer function of a filter can be separated into a real function of f multiplied by a phase factor $e^{j[af + \beta]}$, then this transfer function will exhibit linear phase.

Taking the FIR filter transfer function

$$H(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3} + \ldots + b_{Q-1} z^{-(Q-1)}$$

and replacing z by $e^{j2\pi f}$ to give the frequency response

$$H(f) = b_0 + b_1 e^{-j2\pi f} + b_2 e^{-j2\pi(2f)} + \ldots + b_{Q-1} e^{-j2\pi(Q-1)f}$$

Factoring out the common factors and letting ζ equal $(Q-1)/2$ gives

$$H(f) = e^{-j2\pi\zeta f}\left\{b_0 e^{j2\pi\zeta f} + b_1 e^{j2\pi(\zeta-1)f} + b_2 e^{j2\pi(\zeta-2)f} + \ldots + b_{Q-2} e^{-j2\pi(\zeta-1)f} + b_{Q-1} e^{-j2\pi\zeta f}\right\}$$

Combining the coefficients with complex conjugate phases and placing them together in brackets

$$H(f) = e^{-j2\pi\zeta f}\left\{\left[b_0 e^{j2\pi\zeta} + b_{Q-1} e^{-j2\pi\zeta f}\right] + \left[b_1 e^{j2\pi(\zeta-1)f} + b_{Q-2} e^{-j2\pi(\zeta-1)f}\right] + \left[b_2 e^{j2\pi(\zeta-2)f} + b_{Q-3} e^{-j2\pi(\zeta-2)f}\right] + \ldots\right\}$$

If each pair of coefficients inside the brackets is set equal as follows:

$$b_0 = b_{Q-1}$$
$$b_1 = b_{Q-2}$$
$$b_2 = b_{Q-3}, \text{ etc.}$$

Each term in brackets becomes a cosine function, and the linear phase relationship is achieved. This is a common characteristic of FIR filter coefficients.

1.3.3 IIR Filters

Repeating the general equation for IIR filters

$$y(n) = \sum_{q=0}^{Q-1} b_q x(n-q) - \sum_{p=1}^{P-1} a_p y(n-p)$$

The z-transform of the transfer function of an IIR filter is

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{q=0}^{Q-1} b_q z^{-q}}{1 + \sum_{p=1}^{P-1} a_p z^{-p}}$$

No simple relationship exists between the coefficients of the IIR filter and the impulse response sequence such as that existing in the FIR case. Also, obtaining linear phase IIR filters is not a straightforward coefficient relationship as is the case for FIR filters. However, IIR filters have an important advantage over FIR structures: In general, IIR filters require less coefficients to approximate a given filter frequency response than do FIR filters. This means that results can be computed faster on a general-purpose computer or with less hardware in a special-purpose design.

1.3.4 Example Filter Responses

As an example of the frequency response of an FIR filter with very simple coefficients, take the following MA equation:

$$y(n) = 0.11\,x(n) + 0.22\,x(n-1) + 0.34\,x(n-2) + 0.22\,x(n-3) + 0.11\,x(n-4)$$

One would suspect that this filter would be a lowpass type by inspection of the coefficients, since a constant (DC) value at the input will produce that same value at the output. Also, since all coefficients are positive it will tend to average adjacent values of the signal.

The response of this FIR filter is shown in Figure 1.16. It is indeed lowpass and the nulls in the stopband are characteristic of discrete time filters in general. As an example of the simplest IIR filter, take the following AR equation:

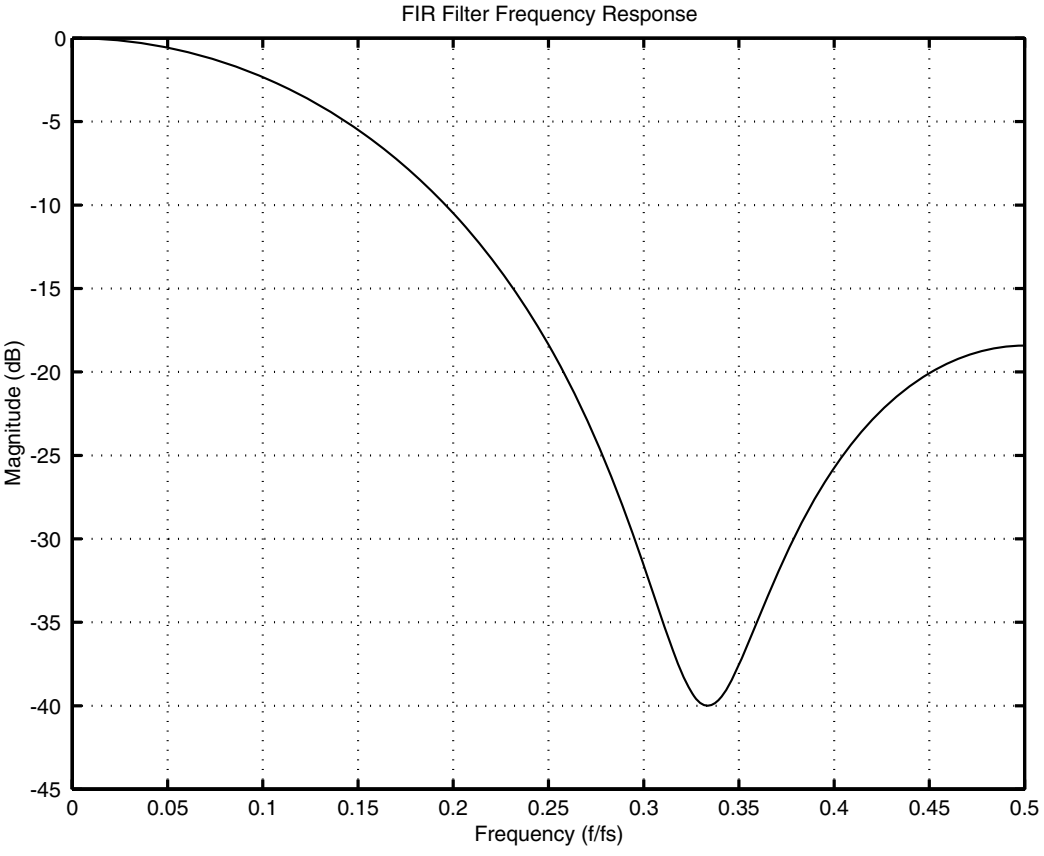


FIGURE 1.16 FIR lowpass frequency response.

$$y(n) = x(n) + y(n-1)$$

Some contemplation of this filter's response to some simple inputs (like constant values, 0, 1, and so on) will lead to the conclusion that it is an integrator. For zero input, the output holds at a constant value forever. For any constant positive input the output grows linearly with time. For any constant negative input the output decreases linearly with time. Note that for a zero frequency input (a DC constant input) the gain of the filter is infinite, which is a result of the filter being unstable. A more useful IIR lowpass filter can be obtained by changing the sign and adding two constants as follows:

$$y(n) \;=\; 0.2 \cdot [x(n) \;+\; x(n-1)] - 0.6 \cdot y(n-1)$$

The frequency response of this filter is shown in Figure 1.17. Figure 1.18 shows the step response of the FIR and IIR lowpass filters. Note that the IIR filter does not reach the step input value of 1.0 (after 10 samples it is at 0.9952), while the FIR filter output reaches exactly 1.0 in 5 samples.

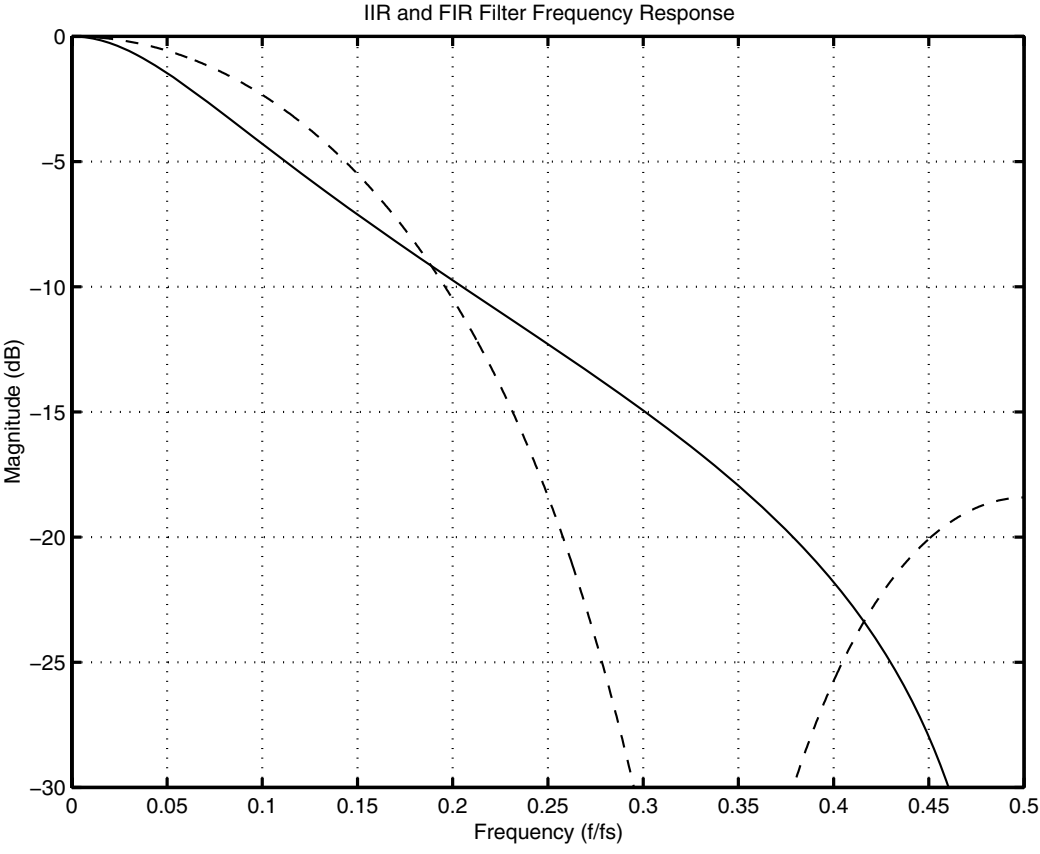


FIGURE 1.17 Example IIR lowpass frequency response (solid line) compared to 5-tap FIR filter frequency response (dashed line) .

1.3.5 Filter Specifications

As mentioned previously, filters are generally specified by their performance in the frequency domain, both amplitude and phase response as a function of frequency. Figure 1.19 shows a lowpass filter magnitude response characteristic. The filter gain has been normalized to be roughly 1.0 at low frequencies. The figure illustrates the most important terms associated with filter specifications.

The region where the filter allows the input signal to pass to the output with little or no attenuation is called the *passband.* In a lowpass filter, the passband extends from frequency f = 0 to the start of the transition band, marked as frequency f_p in Figure 1.19. The *transition band* is that region where the filter smoothly changes from passing the signal to stopping signal. The end of the transition band occurs at the stopband frequency, f_s. The *stopband* is the range of frequencies over which the filter is specified to

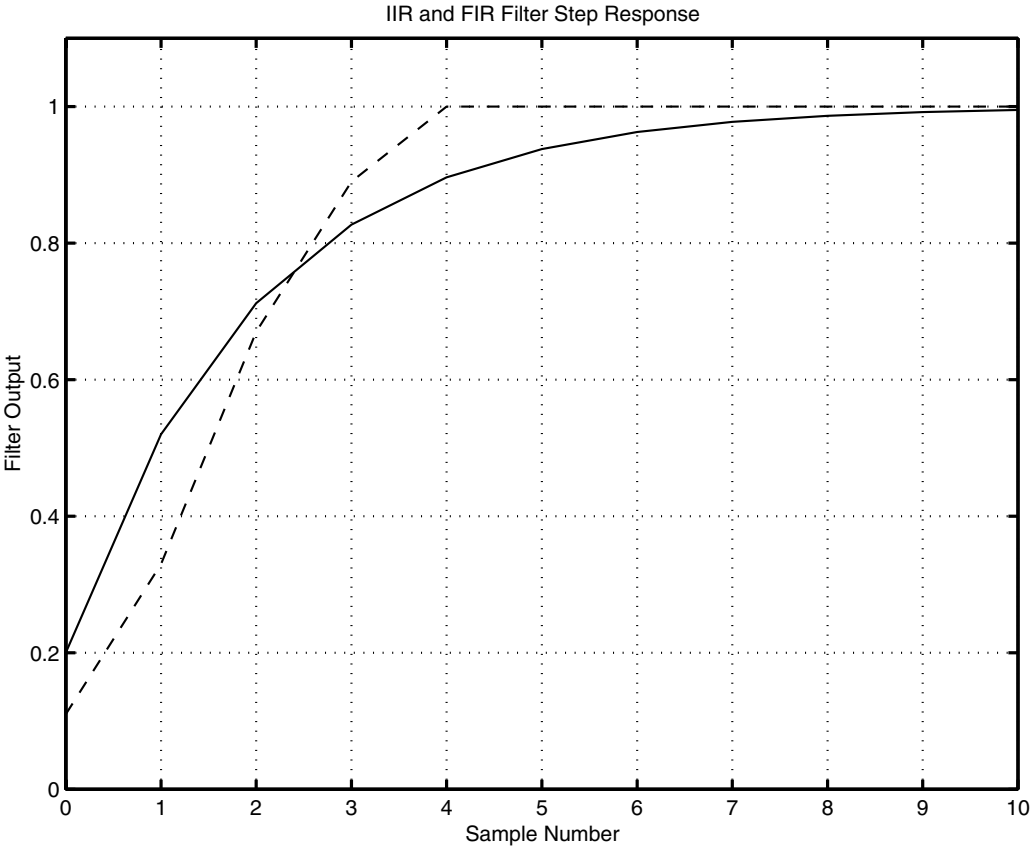


FIGURE 1.18 Step responses of example IIR lowpass filter (solid line) compared to 5-tap FIR lowpass response (dashed line).

attenuate the signal by a given factor. Typically, a filter will be specified by the following parameters:

1. Passband ripple—2δ in the figure.
2. Stopband attenuation—$1/\lambda$.
3. Transition start and stop frequencies—f_p and f_s.
4. Cutoff frequency—f_p. The frequency at which the filter gain is some given factor lower than the nominal passband gain. This may be −1 dB, −3 dB, or another gain value close to the passband gain.

Computer programs that calculate filter coefficients from frequency domain magnitude response parameters use the above list or some variation as the program input.

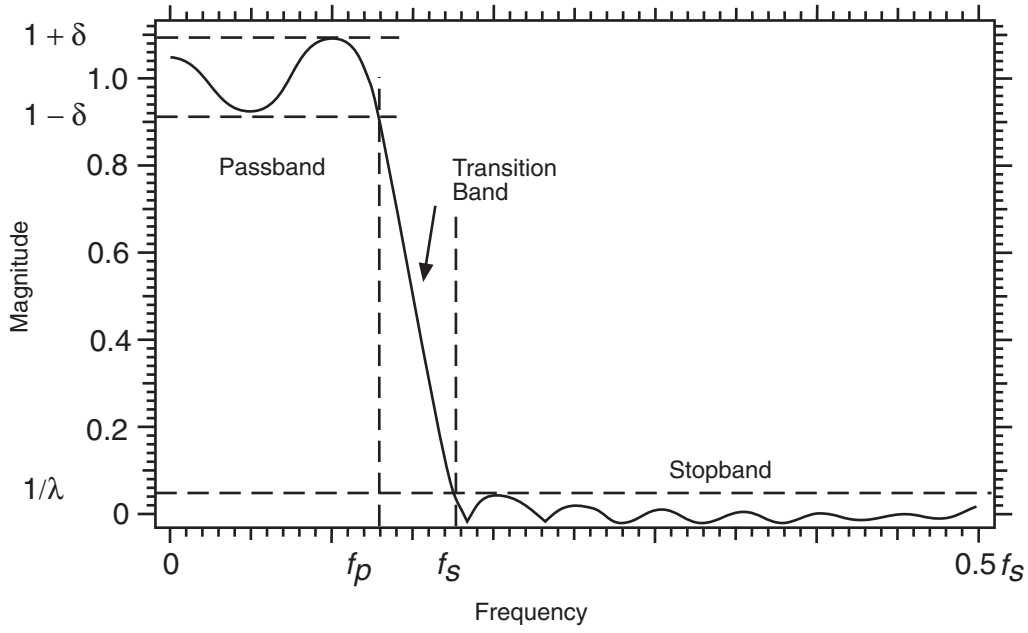


FIGURE 1.19 Magnitude response of normalized lowpass filter.

1.3.6 Filter Structures

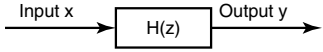
Given the FIR or IIR equation for a filter, there are a number of implementation structures that can be used. Each structure, although mathematically equivalent, may produce different results due to numerical inaccuracies in the computer or special-purpose hardware used.

Figure 1.20 shows three structures for filter implementation. The first [Figure 1.20(a)] is a direct-form implementation of the transfer function. This structure uses the z-transform equation of the filter transfer function and implements each delay and coefficient multiplication directly.

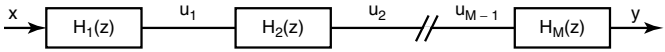
The direct form of the filter can be converted to a cascade form by factoring the transfer function into a set of functions whose product is the overall transfer function [Figure 1.20(b)]. In a similar manner, a partial fraction expansion can be performed on the transfer function to yield a set of functions whose sum is equal to the overall transfer function. This partial fraction expansion leads to the parallel form of the digital filter [Figure 1.20(c)].

1.4 THE DISCRETE FOURIER TRANSFORM

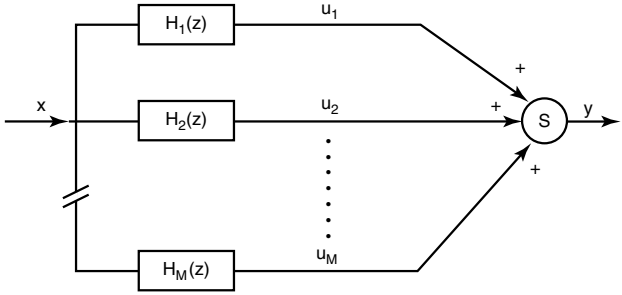
So far, we have made use of the *Fourier transform* several times in our development of the characteristics of sequences and linear operators. The Fourier transform of a causal sequence is



(a) Direct form transfer function.



(b) Cascade form transfer function.



(c) Parallel form transfer function

FIGURE 1.20 Digital filter structures. Adapted from S. Stearns and R. David, *Signal Processing Algorithms,* (Prentice Hall, Englewood Cliffs, NJ, 1988, p. 100).

$$\mathcal{F}\{x(n)\} = X(f) = \sum_{n=0}^{\infty} x(n)e^{-j2\pi fn} \tag{1.65}$$

where the sample time period has been normalized to 1 ($T = 1$). If the sequence is of limited duration (as must be true to be of use in a computer), then

$$X(f) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi fn} \tag{1.66}$$

where the sampled time domain waveform is N samples long. The inverse Fourier transform is

$$\mathcal{F}^{-1}\{X(f)\} = x(n) = \int_{-1/2}^{1/2} X(f)e^{-j2\pi fn}df \tag{1.67}$$

since $X(f)$ is periodic with period $1/T = 1$, the integral can be taken over any full period. Therefore:

$$x(n) = \int_{0}^{1} X(f)e^{-j2\pi fn}df \tag{1.68}$$

1.4.1 Form of the DFT

These representations for the Fourier transform are accurate, but they have a major drawback for digital applications—the frequency variable is continuous, not discrete. To overcome this problem we must approximate both the time and frequency representations of the signal.

To create a *discrete Fourier transform* we must use a sampled version of the frequency waveform. This sampling in the frequency domain is equivalent to convolution in the time domain with the following time waveform:

$$h_1(t) = \sum_{r=-\infty}^{\infty} d(t - rT)$$

This creates duplicates of the sampled time domain waveform that repeat with period T. This T is equal to the T used above in the time domain sequence. Next, by using the same number of samples in one period of the repeating frequency domain waveform as in one period of the time domain waveform, a DFT pair is obtained that is a good approximation to the continuous variable Fourier transform pair. The forward discrete Fourier transform is

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N} \tag{1.69}$$

and the inverse discrete Fourier transform is

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{-j2\pi kn/N} \tag{1.70}$$

For a complete development of the DFT by both graphical and theoretical means, see the text by Brigham (Chapter 6).

1.4.2 Properties of the DFT

This section describes some of the properties of the DFT. The corresponding paragraph numbers in the book *The Fast Fourier Transform* by Brigham are indicated. Due to the sampling theorem, it is clear that no frequency higher than $1/2T$ can be represented by $X(k)$. However, the values of k extend to $N - 1$, which corresponds to a frequency nearly equal to the sampling frequency $1/T$. This means that for a real sequence the values of k from $N/2$ to $N - 1$ are aliased and, in fact, the amplitudes of these values of $X(k)$ are

$$|X(k)| = |X(N-k)|, \text{ for } k = N/2 \text{ to } N-1 \tag{1.71}$$

This corresponds to Properties 8-11 and 8-14 in Brigham.

The DFT is a linear transform as is the z-transform so that the following relationships hold:

If

$$x(n) = \alpha\ a(n) + b\ b(n)$$

where α and ß are constants then

$$X(k) = \alpha\ A(k) + b\ B(k)$$

where $A(k)$ and $B(k)$ are the DFTs of the time functions $a(n)$ and $b(n)$, respectively. This corresponds to Property 8-1 in Brigham.

The DFT also displays a similar attribute under time shifting as the z-transform. If $X(k)$ is the DFT of $x(n)$, then

$$DFT\{x(n-p)\} = \sum_{n=0}^{N-1} x(n-p)e^{-j2kn/N}$$

Now define a new variable $m = r - p$ so that $n = m + p$. This gives

$$DFT\{x(n-p)\} = \sum_{m=-p}^{m=N-1-p} x(m)e^{-j\pi km/N}e^{-j2\pi kp/N}$$

which is equivalent to the following:

$$DFT\{x(n-p)\} = e^{-j2\pi kp/N}X(k) \tag{1.72}$$

This corresponds to Property 8-5 in Brigham. Remember that for the DFT we have assumed that the sequence $x(m)$ goes on forever, repeating its values based on the period $n = 0$ to N - 1. So the meaning of the negative time arguments is simply that

$$x(-p) = x(N-p), \text{ for } p = 0 \text{ to } N-1$$

1.4.3 Power Spectrum

The DFT is often used as an analysis tool for determining the spectra of input sequences. Most often the amplitude of a particular frequency component in the input signal is desired. The DFT can be broken into amplitude and phase components as follows:

$$X(f) = X_{real}(f) + j\ X_{imag}(f) \tag{1.73}$$

$$\begin{aligned} X(f) &= |\ X(f)|e^{j\theta(f)} \\ \text{where } |\ X(f)| &= \sqrt{X_{\text{real}}^2 + X_{\text{imag}}^2} \\ \text{and } \theta(f) &= \tan^{-1}\left[\frac{X_{\text{imag}}}{X_{\text{real}}}\right] \end{aligned} \tag{1.74}$$

If we are interested in the power spectrum of the signal, then we take the signal times its conjugate as follows:

$$X(k)X^*(k) = |X(k)|^2 = X_{real}^2 + X_{imag}^2 \tag{1.75}$$

There are some problems with using the DFT as a spectrum analysis tool, however. The problem of interest here concerns the assumption we made in deriving the DFT that the sequence was a single period of a periodically repeating waveform. Figure 1.21 illustrates this. As can be seen from the picture, for almost all sequences there will be a discontinuity in the time waveform at the boundaries between these pseudoperiods. This discontinuity will result in very high frequency components in the resulting waveform. Since these components can be much higher than the sampling theorem limit of $1/2T$ (or half the sampling frequency), they will be aliased into the middle of the spectrum developed by the DFT.

The technique used to overcome this difficulty is called *windowing*. The problem to be overcome is the possible discontinuity at the edges of each period of the waveform. Since for a general-purpose DFT algorithm there is no way to know the degree of discontinuity at the boundaries, the windowing technique simply reduces the sequence amplitude at the boundaries. It does this in a gradual and smooth manner so that no new discontinuities are produced and the result is a substantial reduction in the aliased frequency components. This improvement does not come without a cost. Because the window is modifying the sequence before a DFT is performed, some reduction in the fidelity of the spectral representation must be expected. The result is somewhat reduced resolution of closely spaced frequency components. The best windows achieve the maximum reduction of *spurious* (or aliased) signals with the minimum degradation of spectral resolution.

There are a variety of windows, but they all work essentially the same way: Attenuate the sequence elements near the boundaries (near $n = 0$ and $n = N - 1$) and compensate by increasing the values that are far away from the boundaries. Each window has its own individual transition from the center region to the outer elements. For a comparison of window performance, see Chapter 5, Section 5.8.1.

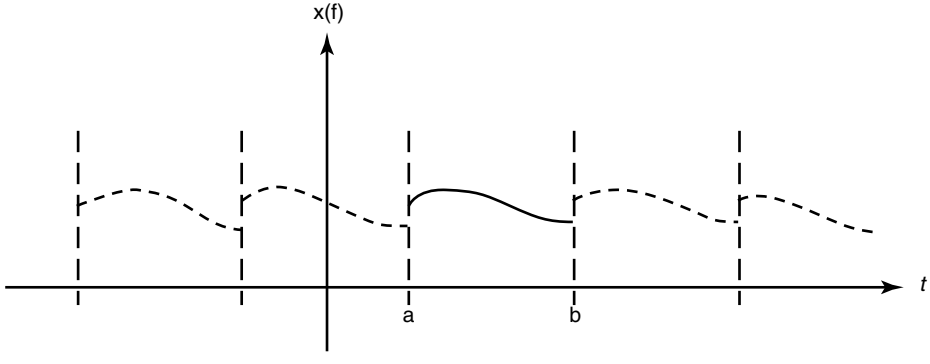


FIGURE 1.21 Making a periodic waveform from a segment.

1.4.4 Averaged Periodograms

Because signals are always associated with noise—either due to some physical attribute of the signal generator or external noise picked up by the signal source—the DFT of a single sequence from a continuous time process is often not a good indication of the true spectrum of the signal. The solution to this dilemma is to take multiple DFTs from successive sequences from the same signal source and take the time average of the power spectrum. If a new DFT is taken, each $N\,T$ seconds and successive DFTs are labeled with superscripts

$$\text{Power Spectrum} = \sum_{i=0}^{M-1}\left[(X_{\text{real}}^{i})^2 + (X_{\text{imag}}^{i})^2\right] \tag{1.76}$$

Clearly, the spectrum of the signal cannot be allowed to change significantly during the interval $t = 0$ to $t = \text{M}$ ($N\,T$).

1.4.5 The Fast Fourier Transform

The *fast Fourier transform* (or FFT) is a very efficient algorithm for computing the DFT of a sequence. It takes advantage of the fact that many computations are repeated in the DFT due to the periodic nature of the discrete Fourier kernel: $e^{-j2\pi kn/N}$. The form of the DFT is

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N} \tag{1.77}$$

By letting $W^{nk} = e^{-j2\pi kn/N}$ we may rewrite Equation 1.77 as

$$X(k) = \sum_{n=0}^{N-1} x(n)W^{nk} \tag{1.78}$$

Now, $W^{(N\,+\,qN)\,(k\,+\,rN)} = W^{nk}$ for all q, r, which are integers due to the periodicity of the Fourier kernel.

We next break the DFT into two parts:

$$X(k) = \sum_{n=0}^{N/2-1} x(2n)W_N^{2nk} + \sum_{n=0}^{N/2-1} x(2n+1)W_N^{(2n+1)k} \tag{1.79}$$

where the subscript N on the Fourier kernel represents the size of the sequence.

If we represent the even elements of the sequence $x(n)$ by x_{ev} and the odd elements by x_{od}, then the equation can be rewritten

$$X(k) = \sum_{n=0}^{N/2-1} x_{ev}(n) W_{N/2}^{nk} + W_{N/2}^{k} \sum_{n=0}^{N/2-1} x_{od}(n) W_{N/2}^{nk} \tag{1.80}$$

We now have two expressions in the form of DFTs so we can write

$$X(k) = X_{ev}(n) + W_{N/2}^{k} X_{od}(n) \tag{1.81}$$

Notice that only DFTs of $N/2$ points need be calculated to find the value of $X(k)$. Since the index k must go to $N-1$, however, the periodic property of the even and odd DFTs is used. In other words:

$$X_{ev}(k) = X_{ev}(k - \tfrac{N}{2}) \qquad \text{for } \tfrac{N}{2} \le k \le N-1 \tag{1.82}$$

The process of dividing the resulting DFTs into even and odd halves can be repeated until one is left with only two point DFTs to evaluate:

$$\begin{aligned} \Lambda(k) &= \lambda(0) + \lambda(1) e^{-j2\pi k/2} && \text{for all } k \\ &= \lambda(0) + \lambda(1) && \text{for } k \text{ even} \\ &= \lambda(0) - \lambda(1) && \text{for } k \text{ odd} \end{aligned}$$

So, for 2-point DFTs no multiplication is required, only additions and subtractions. To compute the complete DFT still requires multiplication of the individual 2-point DFTs by appropriate factors of W ranging from W^0 to $W^{N/2-1}$. Figure 1.22 shows a flow graph of a complete 32-point FFT. We can calculate the savings in computation due to the FFT algorithm as follows.

For the original DFT, N complex multiplications are required for each of N values of k. Also, N - 1 additions are required for each k.

In an FFT each function of the form

$$\lambda(0) \pm W^P \lambda(1)$$

(called a *butterfly* due to its flow graph shape) requires one multiplication and two additions. From the flow graph in Figure 1.22 we can generalize:

$$\text{Number of butterflies} = \frac{N}{2} log_2(N)$$

This is because there are $N/2$ rows of butterflies (since each butterfly has two inputs) and there are $\log2(N)$ columns of butterflies.

Table 1.1 gives a listing of additions and multiplications for various sizes of FFTs and DFTs. The dramatic savings in time for larger DFTs provided in the FFT has made this method of spectral analysis practical in many cases where a straight DFT computation would be much too time-consuming. Also, the FFT can be used for performing operations in the transform domain, which would require much more time-consuming computations in the time domain.

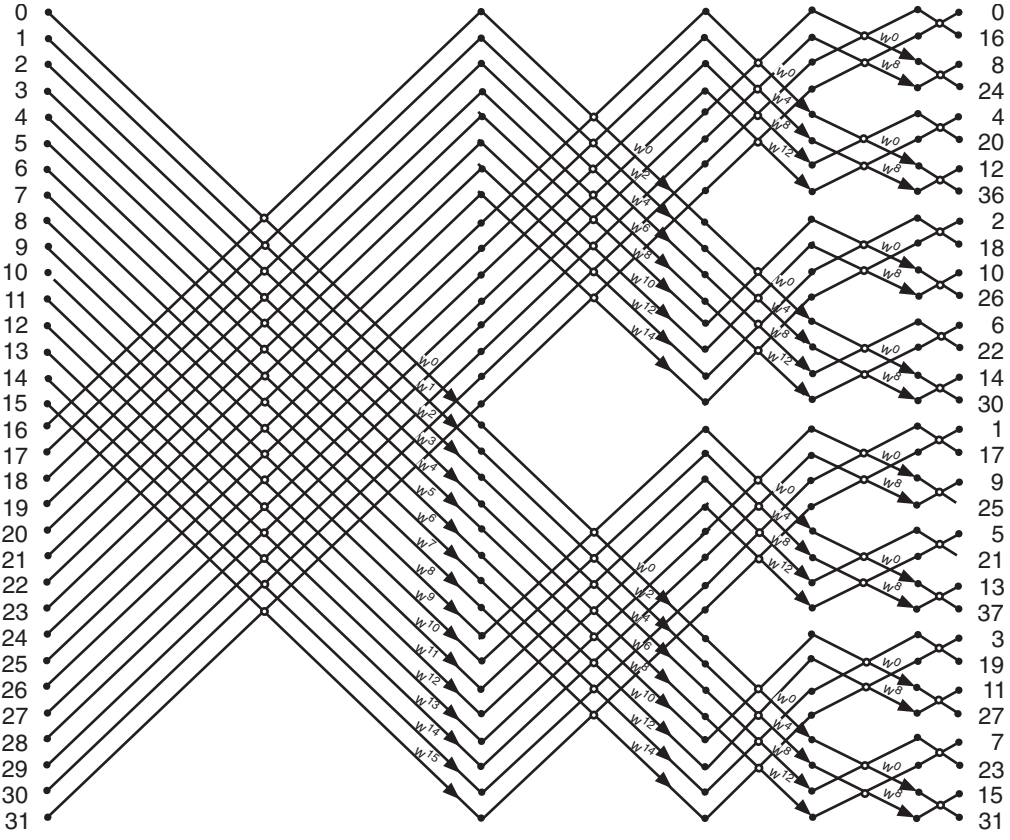


FIGURE 1.22 32-Point, radix 2, in-place FFT. Adapted from L. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, (Prentice Hall, Engelwood Cliffs, NJ, 1975, p. 380).

1.4.6 FFT Example Result

In order to help the reader gain more understanding of spectrum analysis with the FFT a simple example is presented here. An input signal to a 16-point FFT processor is as follows:

$$x(n) = cos\left[2\pi\,(4n/16)\right]$$

The argument of the cosine has been written in an unusual way to emphasize the frequency of the waveform when processed by a 16-point FFT. The amplitude of this signal is 1.0, and it is clearly a real signal, the imaginary component having zero amplitude. Figure 1.23 shows the 16 samples that comprise $x(0)$ to $x(15)$.

TABLE 1.1 Comparison of Number of Butterfly Operations in the DFT and FFT (Each Operation Is One Complex Multiply/Accumulate Calculation)

Transform Length (N)	DFT Operations (N^2)	FFT Operations $N\text{LOG}_2(N)$
8	64	24
16	256	64
32	1024	160
64	4096	384
128	16384	896
256	65536	1024
512	262144	4608
1024	1048576	10240
2048	4194304	22528

With this input a 16-point FFT will produce a very simple output. This output is shown in Figure 1.24. It is a spike at $k = 4$ of amplitude 0.5 and a spike at $k = 12$ of amplitude –0.5. We can see why this is so by referring back to the continuous Fourier transform. For a cosine waveform of arbitrary frequency, the Fourier transform is

$$X(f) = \int_{-\infty}^{+\infty} \cos(2\pi f_0 t) e^{-j2\pi ft} dt$$

Representing the cosine by exponentials

$$X(f) = \frac{1}{2}\int_{-\infty}^{+\infty} e^{j2\pi(f_0 - f)t} dt - \frac{1}{2}\int_{-\infty}^{+\infty} e^{-j2\pi(f_0 + f)t} dt$$

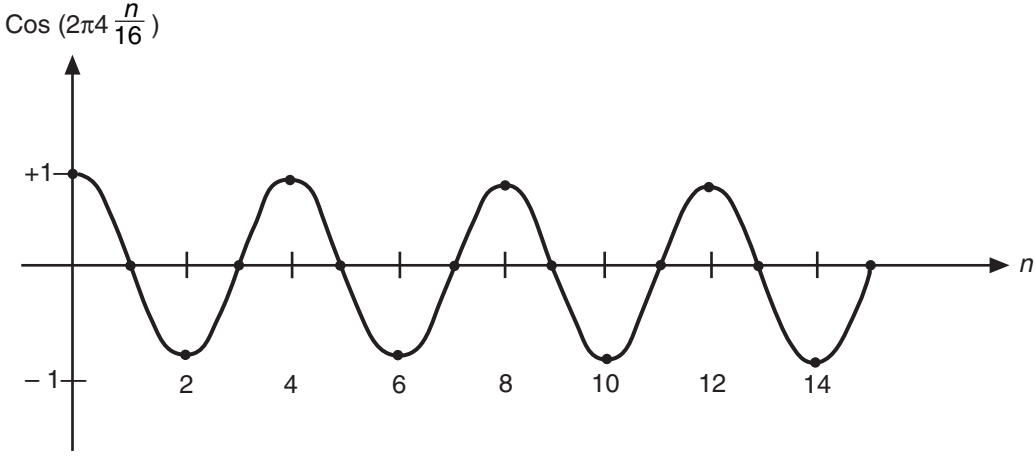


FIGURE 1.23 Input to 16-point FFT.

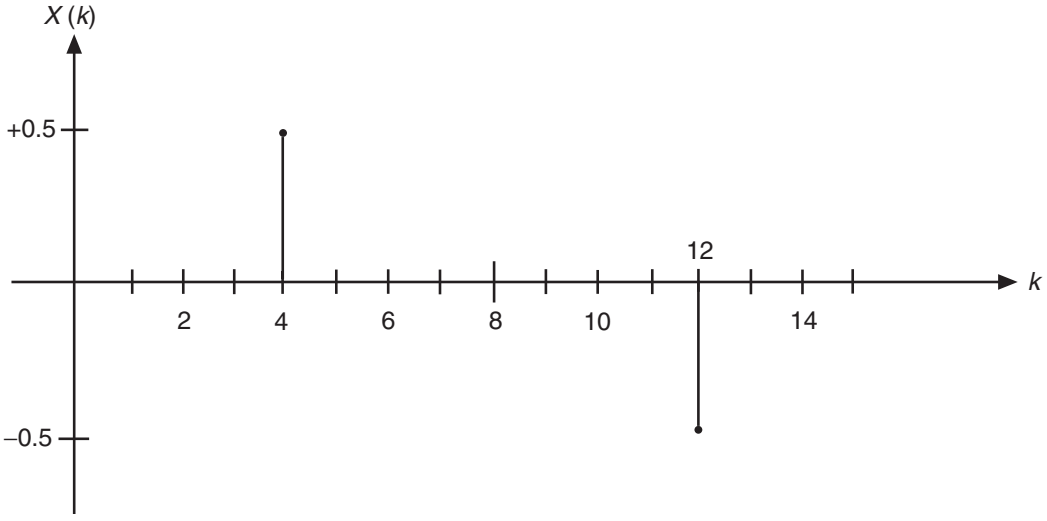


FIGURE 1.24 Output of 16-point FFT.

It can be shown that the integrand in the two integrals above integrates to 0 unless the argument of the exponential is 0. If the argument of the exponential is zero, the result is two infinite spikes, one at $f = f_0$ and the other at $f = -f_0$. These are delta functions in the frequency domain.

Based on these results, and remembering that the impulse sequence is the digital analog of the delta function, the results for the FFT seem more plausible. It is still left to explain why $k = 12$ should be equivalent to $f = -f_0$. Referring back to the development of the DFT, it was necessary at one point for the frequency spectrum to become periodic with period f_s. Also, in the DFT only positive indices are used. Combining these two facts one can obtain the results shown in Figure 1.24.

1.5 NONLINEAR OPERATORS

Most of this book is devoted to linear operators and linear-signal processing because these are the most commonly used techniques in DSP. However, there are several nonlinear operators that are very useful in DSP, and this section introduces these operators.

The nonlinear operators discussed in this section can be grouped into two separate classes: those that compress or clip the input to derive the output sequence (Section 1.5.1) and those that derive the output values from the input by a sorting or searching algorithm (Section 1.5.3).

1.5.1 Clipping and Compression

There is often a need to reduce the number of significant bits in a quantized sequence. This is sometimes done by truncation of the least significant bits. This process is advantageous because it is linear: The quantization error is increased uniformly over the entire range of values of the sequence. There are many applications, however, where the need for accuracy in quantization is considerably less at high-signal values than at low-signal values. This is true, for example, in telephone voice communications, where the human ear's ability to differentiate between amplitudes of sound waves decreases with the amplitude of the sound. In these cases, a nonlinear function is applied to the signal, and the resulting output range of values is quantized uniformly with the available bits.

This process is illustrated in Figure 1.25. First, the input signal from a 12-bit analog to digital converter (ADC) is shown. The accuracy is 12 bits and the range is 0 to 4.095 volts, so each quantization level represents 1 mv. It is necessary because of some system consideration (such as transmission bandwidth) to reduce the number bits in each word to 8. Figure 1.25 shows that the resulting quantization levels are 16 times as coarse. Figure 1.25 shows the result of applying a linear-logarithmic compression to the input signal. In this type of compression the low-level signals (out to some specified value) are unchanged from the input values. Beginning at a selected level, say, $f_{\text{in}} = a$, a logarithmic function is applied. The form of the function might be

$$f_{out} = a + A\log_{10}(1 + f_{in} - a)$$

so that at $f_{\text{in}} = a$ the output also equals a and A is chosen to place the maximum value of f_{out} at the desired point.

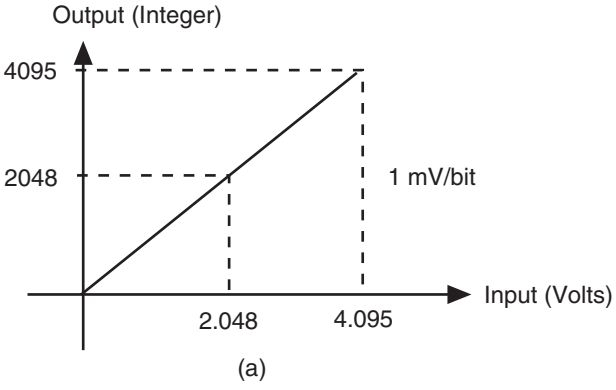
A simpler version of the same process is shown in Figure 1.26. Instead of applying a logarithmic function from the point $f = a$ onward, the output values for $f \geq a$ are all the same. This is an example of clipping. A region of interest is defined, and any values outside the region are given a constant output.

An extreme version of clipping is used in some applications of image processing to produce binary pictures. In this technique a threshold is chosen (usually based on a histogram of the picture elements), and any image element with a value higher than threshold is set to "1" and any element with a value lower than threshold is set to zero. In this way the significant bits are reduced to only one! Pictures properly thresholded can produce excellent outlines of the most interesting objects in the image, which simplifies further processing considerably.

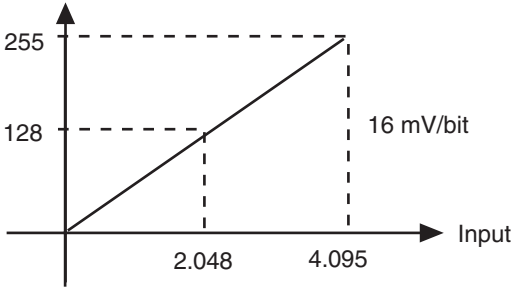
1.5.2 μ-law and A-law Compression

There are two other compression "laws" worth listing because of their use in telephony. They are the μ-*law* and *A-law* conversions. The μ-*law* conversion is defined as follows:

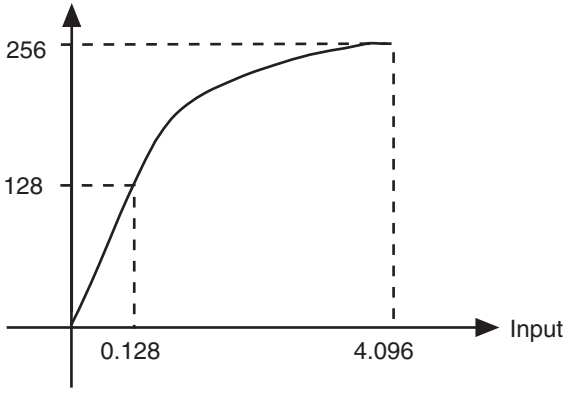
$$f_{\text{out}} = \text{sgn}(f_{\text{in}})\frac{\ln(1 + \mu|f_{\text{in}}|)}{\ln(1 + \mu)} \tag{1.83}$$



(a)



(b)



(c)

FIGURE 1.25 (a) Linear 12-bit ADC. (b) Linear 8-bit ADC. (c) Nonlinear conversion.

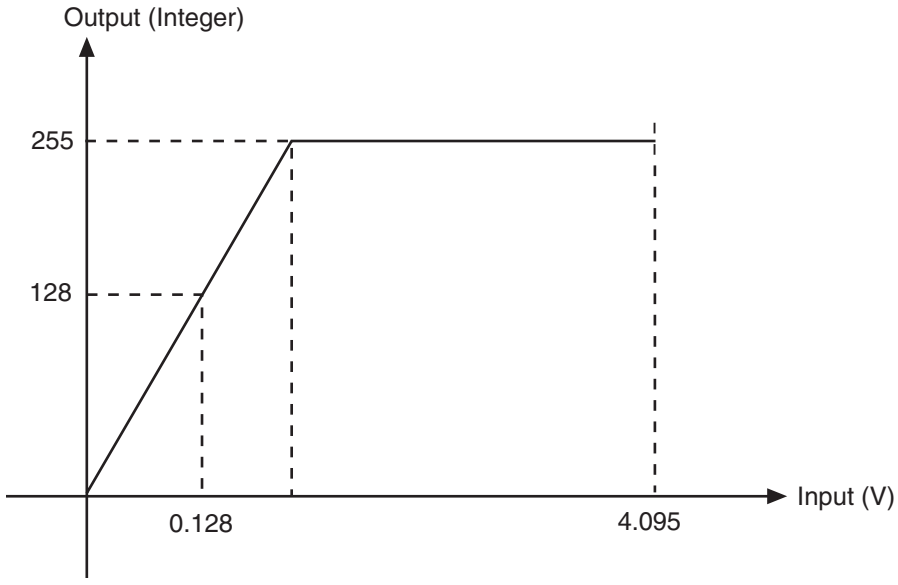


FIGURE 1.26 Clipping to 8 bits.

where sgn() is a function that takes the sign of its argument and μ is the compression parameter (255 for North American telephone transmission). The input value f_{in} must be normalized to lie between −1 and +1. The *A*-law conversion equations are as follows:

$$f_{\text{out}} = \text{sgn}(f_{\text{in}}) \frac{A|f_{\text{in}}|}{1 + \ln(A)}$$

for $|f_{\text{in}}|$ between 0 and $1/A$ and

$$f_{\text{out}} = \text{sgn}(f_{\text{in}}) \frac{1 + \ln(A|f_{\text{in}}|)}{1 + \ln(A)} \tag{1.84}$$

for $|f_{\text{in}}|$ between $1/A$ and 1

In these equations, *A* is the compression parameter (87.6 for European telephone transmission). Both of these compression standards were developed to reduce the bits per sample required to send good-quality speech over digital transmission systems. The speech is normally sampled with 14 bits per sample, and the compressed output is 8 bits.

1.5.3 Filtering by Sorting: Median and Min/Max Filters

Very often the purpose of digital signal processing is to reduce some element of noise that was present in the signal when sampling occurred. There are cases in which the noise exists in a separate part of the frequency spectrum from the signal. In these cases, linear filtering can be quite effective in noise removal, since, with the appropriate filter, the offending section of the spectrum can be attenuated while the signal is relatively uneffected. There are

other cases in which the signal and noise coexist across the spectrum of interest and any filtering that removes noise also degrades the signal. An example of this type of noise is "salt and pepper" noise in images. This type of noise appears as specks (often only one picture element in extent) in the image. The source of this type of noise is often A/D converter problems or digital transmission errors. Bit errors in either of these processes result in impulses in the sequence. These specks or impulse functions spread their energy across a very wide band in the frequency domain. Filtering to remove such noise can result in blurred edges in an image or loss of high-frequency content in a digital sequence.

An effective technique to deal with this type of noise is *median filtering*. Median filtering is performed as follows:

1. For each output point to be generated a window of contiguous data is selected—this could be 15 adjacent time samples in a sequence, or a neighborhood of 9 pixels in an image.
2. The data is sorted so that the order is from highest signal value to lowest value in order.
3. The central value of the sort (i.e., element number 8 in a sort of 15 values or element number 5 in a sort of 9 values) is selected as the median of the data set.
4. The median value of the set is used as the filter output.

Figure 1.27 shows the process of median filtering on a simple data set. One of the results of median filtering that makes it very useful in image processing is that sharp contrast changes in the image are preserved (not blurred as in averaging) while the single spikes that compose "salt and pepper" noise are removed.

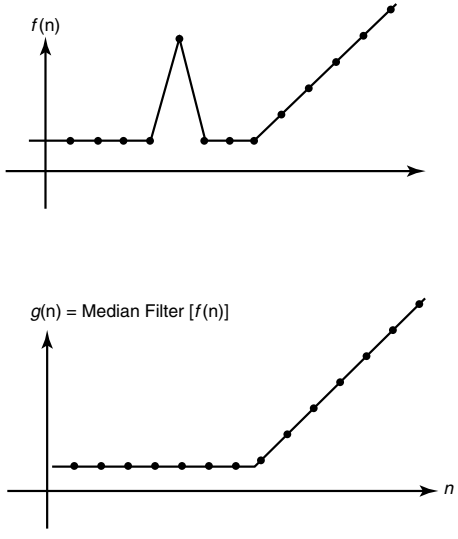


FIGURE 1.27 Median filtering of waveform with a spike.

The procedure outlined above is actually a general one-for-all form of sort-type filtering. Once the data is sorted, any value can be chosen for the filter output. The type of filter is highly dependent on the type of sort. In images, max or min values are often used in place of the median. When max values are used, the brightest areas of the picture tend to expand. The filter can be applied repetitively to slowly "grow" the brighter portions of the picture. If the darker portions are of more interest then one would select the min and apply a repetitive min filter. A repetitive filter with a sorting filter that selects the mode of the data set (rather than the median) tends to produce a mottled effect: Adjacent regions tend to move apart in value.

A more sophisticated technique than straight median filtering is *conditional median filtering*. This algorithm is implemented by the following steps:

1. An appropriately sized window is selected.
2. The data is sorted from highest to lowest.
3. The median is determined.
4. If the absolute value of the difference between the median and the input signal value is greater than some threshold, then the input sample is replaced by the median. Otherwise, the input sample itself is used as the output.

This technique avoids unnecessary corruption of the signal but retains the impulse-removing properties of the straight median filter.

1.6 LINEAR ALGEBRA: MATRICES AND VECTORS

This section presents some of the most useful elements of *matrix* and *vector theory* for DSP. The subject is a very broad one, and only a short overview can be presented here. However, because the notation and concepts are so prevalent in DSP, some introduction is in order.

1.6.1 Vectors

If a sequence of values $x(n)$ is defined over the interval $n = 1$ to $n = N$, the sequence can be written as follows:

$$\{x(n)\} = x(1),\ x(2),\ x(3),\ \ldots,\ x(N-2),\ x(N-1),\ x(N)$$

Another notation that is often used for such a sequence is vector notation. In vector notation the sequence would be written

$$\{x(n)\} = \mathbf{x} = \begin{bmatrix} x(1) \\ x(2) \\ x(3) \\ \vdots \\ x(n) \end{bmatrix} \tag{1.85}$$

The entire sequence is represented by the symbol **x**. This compactness of notation is one of the benefits of vector and matrix theory.

Whenever a vector is not specifically written out, it can be assumed to be arranged in a column as in Equation 1.85. There are times when a row orientation of a vector is preferred, and this is signified by the superscript ^{T} standing for transpose. The transpose of the vector **x** is

$$\mathbf{x}^T = [x(1),\ x(2),\ x(3),\ \ldots,\ x(N)] \tag{1.86}$$

Filtering or windowing calculations are often presented with equations such as

$$y(n) = \sum_{m=1}^{M} w_m x(n-m-1) \tag{1.87}$$

In vector notation this is written as the product of a row vector (or a vector transpose) times a column vector:

$$\mathbf{y} = \mathbf{w}^{\mathrm{T}}\mathbf{p}$$

$$\text{where } \mathbf{p} = \begin{bmatrix} x(n) \\ x(n-1) \\ \vdots \\ x(n-\mathrm{N}) \end{bmatrix}$$

Linear operators can be written as the product of a matrix and vector. Matrix-vector multiplication of **A x** is defined as follows:

$$y_i = \sum_{j=0}^{J} a_{ij} x_j \tag{1.88}$$

where the first subscript of each variable refers to the row number and the second subscript (in the case of matrices) refers to the column number. In the case of convolutional operators such as FIR filters, the rows of the matrix have a very definite relationship to each other. In fact each row is simply a shifted version of the row just above it.

$$\begin{bmatrix} y(n) \\ y(n-1) \\ y(n-2) \\ \vdots \\ y(n-N+1) \end{bmatrix} = \begin{bmatrix} h(0) & h(1) & h(2)\ldots h(N-1) & 0 & 0\ldots 0 \\ 0 & h(0) & h(1)\ldots h(N-2) & h(N-1) & 0\ldots 0 \\ 0 & 0 & h(0)\ldots h(N-3) & h(N-2) & \ldots 0 \\ \vdots & \vdots & \vdots\vdots & \vdots & \vdots \\ 0 & 0 & 0\ldots\ldots & & h(N-1) \end{bmatrix} \begin{bmatrix} x(n) \\ x(n-1) \\ x(n-2) \\ \vdots \\ x(n-N+1) \end{bmatrix}$$

The DFT can also be presented in matrix form, since it is simply another instance of a linear operator. The matrix is made up of the *twiddle factors,* or powers of the exponential kernel of the DFT.

1.6.2 Properties of Matrix Mathematics

Matrix-vector multiplication was described above, and some examples of the compactness of the notation were given. This section gives a more complete introduction to the rules of matrix algebra and is written to be used as a concise reference for the most commonly used theorems. The properties are all written for matrices but can be used equally well for vectors considered as the special case of a matrix with one column.

Property 1. If the individual elements of a matrix **A** are represented as

$$a_{ij}$$

where i is the row number and j the column number of the element, and **A** is an M-row by N-column matrix and **B** is an M x N matrix, then matrix addition is defined as

$$\mathbf{C} = \mathbf{A} + \mathbf{B} \text{ where}$$

$$c_{ij} = a_{ij} + b_{ij}$$

Property 2. If **A** is an M-row by N-column matrix and **B** is an N x Q matrix, then matrix multiplication is defined as

$$\mathbf{C} = \mathbf{AB} \text{ where}$$

$$c_{ij} = \sum_{k=1}^{N} a_{ij} b_{kj} \text{ for } i = 1 \text{ to } M \text{ and } j = 1 \text{ to } Q$$

Property 3. There exists an identity matrix for addition called the **0** matrix. Each element of this matrix equals zero.

Property 4. There exists an identity matrix for multiplication that is generally just called the *identity matrix* and whose symbol is I. This matrix is always square (for instance, M x M) and consists of all zero elements except for the main diagonal, which has all elements equal to 1. For any Q x M matrix the following identity holds:

$$\mathbf{AI} = \mathbf{A}$$

For any M x Q matrix the following identity holds:

$$\mathbf{IA} = \mathbf{A}$$

Property 5. Matrix multiplication is associative:

$$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$$

Property 6. Matrix multiplication is distributive:

$$\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$$

Property 7. Matrix multiplication is not commutative: In general, **AB** does not equal **BA.**

Property 8. Matrix **A** is invertible if there exists a matrix **B** such that **AB = I,** the identity matrix and **BA = I.** If there is such a **B** it is called the *inverse* of **A** and denoted by $\mathbf{A}^{-1}$.

Property 9. The transpose of a product of matrices can be written as the product of the transposes in reverse order:

$$(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$$

Property 10. The product of invertible matrices has an inverse given by

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$$

Property 11. The *complex conjugate of a matrix* is a matrix consisting of the complex conjugates of each element in the original matrix. It is usually denoted by a superscript asterisk. Complex conjugate $\{\mathbf{A}\} = \mathbf{A}^*$.

Property 12. The transpose of the complex conjugate of a matrix is called the *Hermitian transpose* and is denoted by a superscript ^{H}.

$$(\mathbf{A}^*)^T = (\mathbf{A}^T)^* = \mathbf{A}^H$$

Property 13. An *orthogonal matrix* is one that has the following characteristics:

$$\mathbf{Q}^T\mathbf{Q} = \mathbf{I},\ \mathbf{QQ}^T = \mathbf{I} \text{ and } \mathbf{Q}^T = \mathbf{Q}^{-1}$$

Property 14. The *length of a vector* is the scalar that is the square root of the product of the vector transpose with the vector itself:

$$\|\mathbf{x}\|^2 = \mathbf{x}^T\mathbf{x}$$

1.7 PROBABILITY AND RANDOM PROCESSES

The signals of interest in most signal processing problems are embedded in an environment of noise and interference. The noise may be due to spurious signals picked up during transmission (interference), to the noise characteristics of the electronics that receives the signal or a number of other sources. To deal effectively with noise in a signal, some

model of the noise or of the signal plus noise must be used. Most often a probabilistic model is used, since the noise is, by nature, unpredictable. This section introduces the concepts of probability and randomness, which are basic to digital signal processing, and gives some examples of the way a composite signal of interest plus noise is modeled.

1.7.1 Basic Probability

Probability begins by defining the probability of an event labeled A as $P(A)$. Event A can be the result of a coin toss, the outcome of a horse race, or any other result of an activity that is not completely predictable. There are three attributes of this probability $P(A)$:

1. $P(A) >= 0$. Any result will either have a positive chance of occurrence or no chance of occurrence.
2. P (All possible outcomes) = 1. Some result among those possible is bound to occur, a probability of 1 being certainty.
3. For $\{Ai\}$, where $(A_i \cap A_i) = 0$, $P(\cup A_i) = \Sigma_i P(A_i)$. For a set of events, $\{Ai\}$, where the events are mutually disjoint (no two can occur as the result of a single trial of the activity), the probability of any one of the events occurring is equal to the sum of their individual probabilities.

With probability defined in this way, the discussion can be extended to joint and conditional probabilities. Joint probability is defined as the probability of occurrence of a specific set of two or more events as the result of a single trial of an activity. For instance, the probability that horse A will finish third and horse B will finish first in a particular horse race is a joint probability. This is written

$$P(A \cap B) = P(A \textit{ and } B) = P(AB) \tag{1.89}$$

Conditional probability is defined as the probability of occurrence of an event A given we know that B has occurred. For example, if the activity is driving to work, then two of the possible events in the set of all possible events are *A:* you arrive at work on time, and *B:* you get a flat tire.

These events are not mutually exclusive. You may leave early enough to fix a flat and arrive on time. But we would associate a lower probability with event A if we knew that event B had occurred. So the probability assigned to event A is conditioned by our knowledge of event B. This is written

$$P(A \textit{ given } B) = P(A|B) \tag{1.90}$$

If this conditional probability, $P(A|B)$, and the probability of B (getting a flat tire at all) are both known, we can calculate the probability of both of these events occurring (joint probability) as

$$P(AB) = P(A|B)P(B) \tag{1.91}$$

So the conditional probability is multiplied by the probability of the condition (event B) to get the joint probability. Another way to write this equation is

$$P(A|B) = \frac{P(AB)}{P(B)} \tag{1.92}$$

This is another way to define conditional probability once joint probability is understood.

Bayes' rule is an often used equation in probability theory and is written in its simplest form

$$P(AB) = P(BA) \tag{1.93}$$

This is satisfying, since nothing in joint probability is affected by the order in which the events are listed. The most often used form of Bayes' rule uses the definition of conditional probability:

$$P(A|B)P(B) = P(B|A)P(A) \tag{1.94}$$

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{1.95}$$

This last equation is often useful when the probabilities of individual events and one of the conditional probabilities are known and we would like to find the other conditional probability.

1.7.2 Random Variables

In signal processing, the probability of a signal taking on a certain value or lying in a certain range of values is often desired. The signal in this case can be thought of as a *random variable* (an element whose set of possible values is the set of outcomes of the activity). For instance, if our random variable is X, we might have the following set of events which could occur:

Event A X takes on the value of 5 ($X = 5$)
Event B $X = 19$
Event C $X = 1.66$
–
–
etc.

This is a useful set of events for discrete variables that can only take on certain specified values. A more practical set of events for continuous variables associates each event with the variable lying within a range of values. We can define a cumulative distribution function for a random variable as follows:

$$F(x) = P(X \leq x) \tag{1.96}$$

This cumulative distribution function then is a monotonically increasing function of the independent variable x and is valid only for the particular random variable, X. Figure 1.28 shows an example of a distribution function for a random variable. If we differentiate $F(x)$ with respect to x, we get the probability density function (PDF) for X, represented as follows:

$$p(x) = \frac{dF(x)}{dx} \tag{1.97}$$

Integrating $p(x)$ gives the distribution function back again as follows:

$$F(x) = \int_{-\infty}^{x} p(\lambda) d\lambda \tag{1.98}$$

Since $F(x)$ is always monotonically increasing, $p(x)$ must be always positive or zero. Figure 1.29 shows the density function for the distribution of Figure 1.28. The utility of these functions can be illustrated by determining the probability that the random variable X lies between a and b. By using probability Property 3 from above

$$P(X \le b) = P(a < X \le b) + P(X \le a) \tag{1.99}$$

This is true because the two conditions on the right-hand side are independent (mutually exclusive) and X must meet one or the other if it meets the condition on the left-hand side. We can rewrite this equation using the definition of the distribution:

$$\begin{aligned} P(a < X \le b) &= F(b) - F(a) \\ &= \int_{a}^{b} p(x) dx \end{aligned} \tag{1.100}$$

In this way, knowing the distribution or the density function allows the calculation of the probability that X lies within any given range.

1.7.3 Mean, Variance, and Gaussian Random Variables

There is an operator in random variable theory called the *expectation operator* usually written E[x]. This expression is pronounced "the expected value of x." The expectation operator extracts from a random variable the value that the variable is most likely to take on. The expected value is sometimes called the *mean, average* or *first moment* of the variable and is calculated from the density function as follows:

$$\mathrm{E}[x] = \int_{-\infty}^{\infty} x p(x) dx \tag{1.101}$$

A typical density function for a random variable is shown in Figure 1.30. The most likely value of variable x is also indicated in the figure. The expected value can be thought of as a "center of gravity" or first moment of the random variable x.

The variance of a random variable is defined as

$$\sigma^2 = \mathrm{Var}\{x\} = \mathrm{E}\left[(x - \mathrm{E}[x])^2\right] \tag{1.102}$$

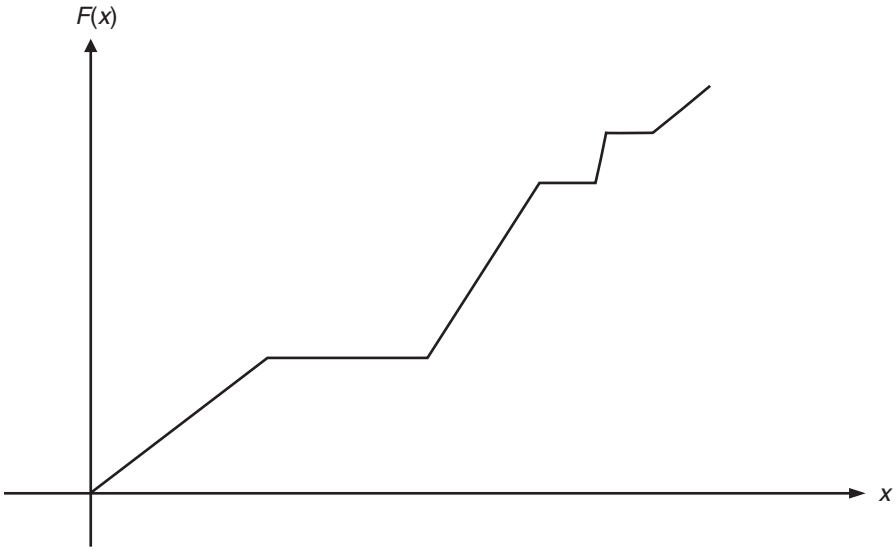


FIGURE 1.28 An example cumulative distribution function (CDF).

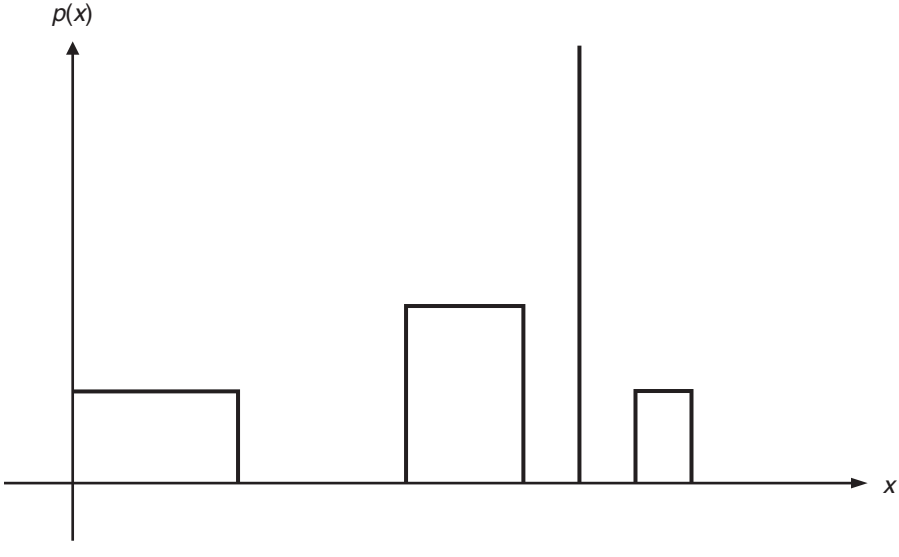


FIGURE 1.29 Density function.

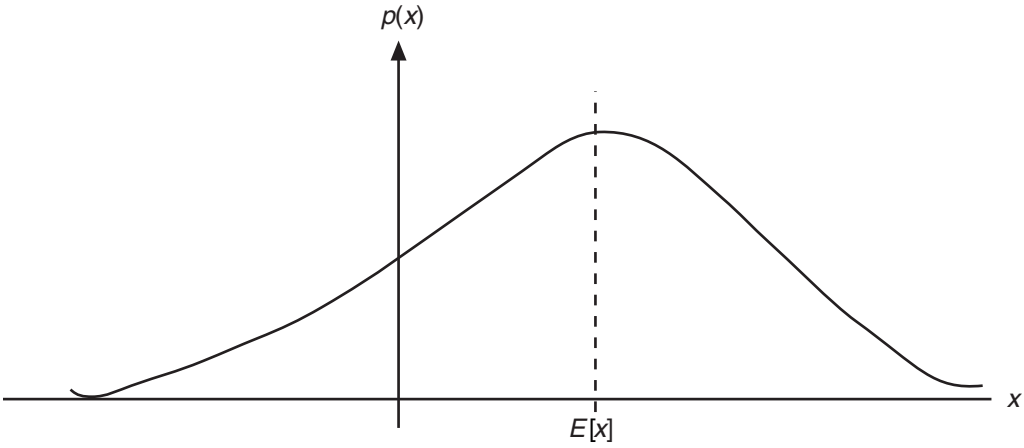


FIGURE 1.30 Continuous PDF showing *E[x].*

where σ is the root mean square value of the variable's difference from the mean. The variance is sometimes called the *mean square value* of *x*.

By extending the use of the expectation operator to joint probability densities, a variable *Y* can be a function of two random variables, *s* and *t,* such that

$$Y = \mathbb{O}\{s, t\}$$

Then the expected value of *Y* will be

$$\mathrm{E}[Y] = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \mathbb{O}\{s, t\}p(s, t)ds\, dt \tag{1.103}$$

where the joint probability density of *s* and *t,* ($p(s,t)$), is required in the equation. We define the correlation of two random variables to be the expected value of their product

$$\mathrm{E}[st\} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} st\ p(s, t)ds\, dt \tag{1.104}$$

This definition will be used in the development of autocorrelation in Section 1.7.5.

There is a set of random variables called *Gaussian random variables* whose density functions have special characteristics that make them particularly easy to analyze. Also, many physical processes give rise to approximately this sort of density function. A *Gaussian density function* has the following form:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] \tag{1.105}$$

where μ is the mean value of *x* and σ^2 is the variance.

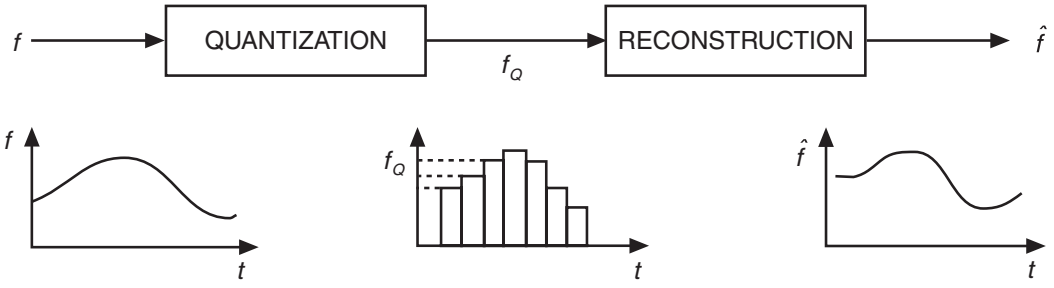


FIGURE 1.31 Quantization and reconstruction of a signal.

1.7.4 Quantization of Sequences

Quantization is to the amplitude domain of a continuous analog signal as sampling is to the time domain. It is the step that allows a continuous amplitude signal to be represented in the discrete amplitude increments available in a digital computer. To analyze the process of quantization it is useful to diagram a system as shown in Figure 1.31. The illustration shows a continuous amplitude input signal, f, that is quantized and then reconstructed in the continuous amplitude domain. The output signal is f. By comparing the input and output of this process the effect of quantization can be illustrated.

The action of the box marked quantization in Figure 1.31 is illustrated in Figure 1.32. A set of decision levels is applied to each input signal, and the two levels that

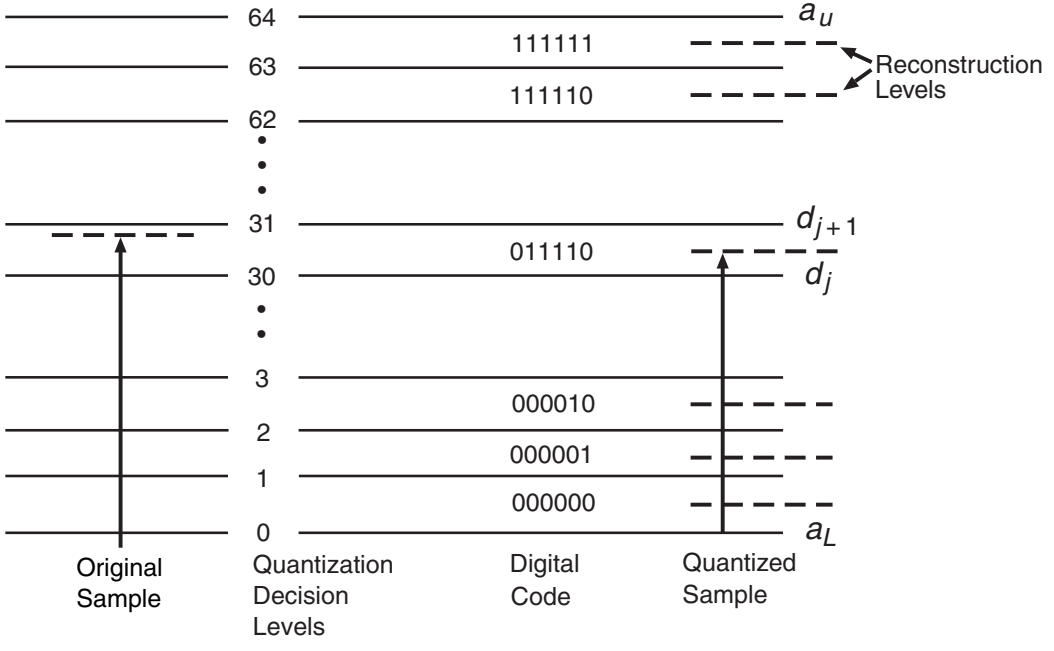


FIGURE 1.32 Quantization operation.

bracket the signal above and below are determined. A digital code is assigned to the region between each levels. In Figure 1.32 the digital code consists of 6 bits and runs from binary 0 to binary 63. The application of these decision levels and the assignment of a code to the input signal sample is the complete process of quantization. Reconstruction of the signal is accomplished by assigning a reconstruction level to each digital code.

The task that remains is to assign actual values to the decision levels and the reconstruction levels. The minimum value of the input signal is labeled a_L, and the maximum value is labeled a_U. If the signal f has a probability density of $p(f)$, then the mean squared error due to the quantization and reconstruction process is

$$\varepsilon = E\left\{(f - \hat{f})^2\right\} = \int_{a_L}^{a_U} (f - \hat{f})^2 \, p(f) df$$

and if the signal range is broken up into the segments between decision levels

$$\varepsilon = E\left\{(f - \hat{f})^2\right\} = \sum_{J=0}^{J-1} \int_{d_j}^{d_{j+1}} (f - r_j)^2 \, p(f) df$$

Numerical solutions can be determined that minimize ε for several common probability densities. The most common assumption is a uniform density ($p(f)$ equals $1/N$ for all values of f, where N is the number of decision intervals. In this case, the decision levels are uniformly spaced throughout the interval and the reconstruction levels are centered between decision levels. This method of quantization is almost universal in commercial analog-to-digital converters. For this common case the error in the analog-to-digital converter output is uniformly distributed from $-\frac{1}{2}$ of the least significant bit to $+\frac{1}{2}$ of the least significant bit. If we assume that the value of the least significant bit is unity, then the mean squared error due to this uniform quantization is given by

$$var\{\varepsilon\} = \int_{-\frac{1}{2}}^{+\frac{1}{2}} (f - \hat{f})^2 \, p(f) df = \int_{-\frac{1}{2}}^{+\frac{1}{2}} f^2 df = \frac{1}{12}$$

since $p(f) = 1$ from -½ to +½. This mean squared error gives the equivalent variance or noise power added to the original continuous analog samples as a result of the uniform quantization. If we further assume that the quantization error can be modeled as a stationary, uncorrelated white noise process (which is a good approximation when the number of quantization levels is greater than 16), then a maximum signal to noise ratio can be defined for a quantization process of B bits (2^B quantization levels) as follows:

$$SNR = 10\log_{10}(V^2/\text{var}\{\varepsilon\}) = 10\log_{10}(12V^2)$$

where V^2 is the total signal power. For example, if a sinusoid is sampled with a peak amplitude of 2^{B-1}, then $V^2 = 2^{2B}/8$, giving the signal-to-noise ratio for a full-scale sinusoid as

$$SNR = 10\log_{10}((1.5)(2^{2B})) = 6.02B + 1.76$$

This value of SNR is often referred to as the theoretical signal-to-noise ratio for a B bit analog-to-digital converter. Because the analog circuits in a practical analog-to-digital converter always add some additional noise, the SNR of a real-world converter is always less than this value.

1.7.5 Random Processes, Autocorrelation, and Spectral Density

A *random process* is a function composed of random variables. An example is the random process $f(t)$. For each value of t, the process $f(t)$ can be considered a random variable. For $t = a$ there is a random variable $f(a)$ that has a probability density, an expected value (or mean), and a variance as defined in Section 1.7.3. In a two-dimensional image, the function would be $f(x,y)$, where x and y are spatial variables. A two-dimensional random process is usually called a *random field.* Each $f(a,b)$ is a random variable.

One of the important aspects of a random process is the way in which the random variables at different points in the process are related to each other. Here we need to extend the concept of joint probability to distribution and density functions. A *joint probability distribution* is defined as

$$F(s, t) = P(S \le s, T \le t) \text{ (where } S \text{ and } T \text{ are some constants)}$$

and the corresponding density function is defined as

$$p(s, t) = \frac{\partial^2 F(s, t)}{\partial s \partial t} \tag{1.106}$$

The integral relationship between distribution and density in this case is

$$F(s, t) = \int_{-\infty}^{s} \int_{-\infty}^{t} p(\alpha, \beta) d\alpha \, d\beta \tag{1.107}$$

We have seen in Section 1.7.3 that the correlation of two random variables is the expected value of their product. The *autocorrelation* of a random process is the expected value of the products of the random variables that make up the process. The symbol for autocorrelation is $R_{\text{ff}}(t_1, t_2)$ for the function $f(t)$ and the definition is

$$R_{ff}(t_1, t_2) = \mathrm{E}\left[f(t_1) f(t_2)\right] \tag{1.108}$$

$$= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \alpha \, \beta \, p_f(\alpha, \beta; t_1, t_2) d\alpha \, d\beta \tag{1.109}$$

where $p_{\text{f}}(\alpha, \beta; t_1, t_2)$ is the joint probability density $f(t_1)$ and $f(t_2)$. By including α and ß in the parentheses, the dependence of p_{f} on these variables is made explicit.

In the general case, the autocorrelation can have different values for each value of t_1 and t_2. However, there is an important special class of random processes called *stationary processes* for which the form of the autocorrelation is somewhat simpler. In station-

ary random processes, the autocorrelation is only a function of the difference between the two time variables. For stationary processes we can then write

$$R_{ff}(t_2 - t_1) = R_{ff}(\xi) = \mathrm{E}\big[f(t-\xi)f(t)\big] \tag{1.110}$$

In Section 1.7.6 the continuous variable theory presented here is extended to discrete variables, and the concept of modeling real-world signals is introduced.

1.7.6 Modeling Real-World Signals With AR Processes

By its nature, a noise process cannot be specified as a function of time in the way a deterministic signal can. We must be satisfied with the probability function and the first and second moments of the process. Although this is only a partial characterization, a considerable amount of analysis can be performed using moment parameters alone. The first moment of a process is simply its average or mean value. In this section, all processes will have zero mean, simplifying the algebra and derivations but providing results for the most common set of processes.

The second is the autocorrelation of the process

$$r(n, n-k) = \mathrm{E}\big[u(n)u^*(n-k)\big], \qquad \text{for } k = 0, \pm 1, \pm 2, \ldots.$$

The processes we will consider are *stationary* to *second order*. This means that the first- and second-order statistics do not change with time. This allows the autocorrelation to be represented by

$$r(n, n-k) = r(k), \qquad \text{for } k = 0, \pm 1, \pm 2, \ldots$$

since it is a function only of the time *difference* between samples and not the time variable itself. In any process, an important member of the set of autocorrelation values is $r(0)$, which is

$$r(0) = \mathrm{E}\big\{u(n)u^*(n)\big\} = \mathrm{E}\Big\{\big|u(n)\big|^2\Big\} \tag{1.111}$$

which is the mean square value of the process. For a zero mean process this is equal to the variance of the signal

$$r(0) = var\{u\} \tag{1.112}$$

If we represent our process by a vector $\mathbf{u}(n)$, where

$$\mathbf{u}(n) = \begin{bmatrix} u(n) \\ u(n-1) \\ u(n-2) \\ \vdots \\ u(n-M+1) \end{bmatrix} \tag{1.113}$$

then the autocorrelation can be represented in matrix form

$$\mathbf{R} = \mathrm{E}\left\{\mathbf{u}(n)\mathbf{u}^H(n)\right\}$$

$$\mathbf{R} = \begin{bmatrix} r(0) & r(1) & r(2)... & r(M-1) \\ r(-1) & r(0) & r(1)... & \vdots \\ r(-2) & r(-1) & r(0)... & \vdots \\ \vdots & \vdots & \vdots & r(-1) \\ \vdots & \vdots & \vdots & r(0) \\ r(-M+1) & r(-M+2) & \ldots\ldots & r(1) \end{bmatrix} \tag{1.114}$$

The second moments of a noise process are important because they are directly related to the power spectrum of the process. The relationship is

$$S(f) = \sum_{k=-M+1}^{M-1} r(k)e^{-j2\pi fk} \tag{1.115}$$

which is the discrete Fourier transform of the autocorrelation of the process (r(k)). Thus, the autocorrelation is the time domain description of the second-order statistics, and the power spectral density, $S(f)$, is the frequency domain representation. This power spectral density can be modified by discrete time filters.

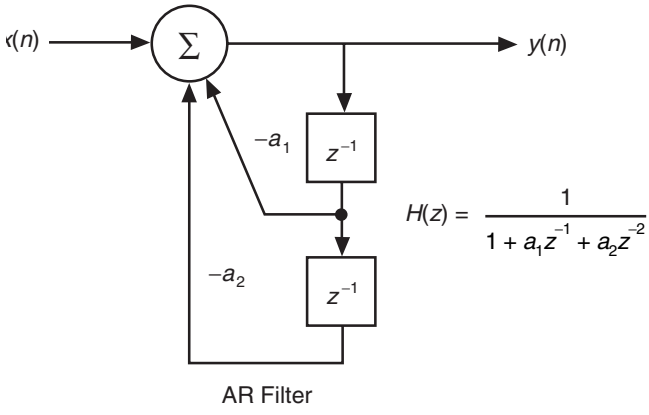
Discrete time filters may be classified as *autoregressive* (AR), *moving average* (MA), or a *combination of the two* (ARMA). Examples of these filter structures and the z-transforms of each of their impulse responses are shown in Figure 1.33. Clearly, the output of a filter has a considerably altered power spectral density when compared with the input. As a result, the second-order statistics, $r(k)$, of the output process will be entirely different from those for the input process. It is theoretically possible to create any arbitrary output stochastic process from an input white noise Gaussian process using a filter of sufficiently high (possibly infinite) order.

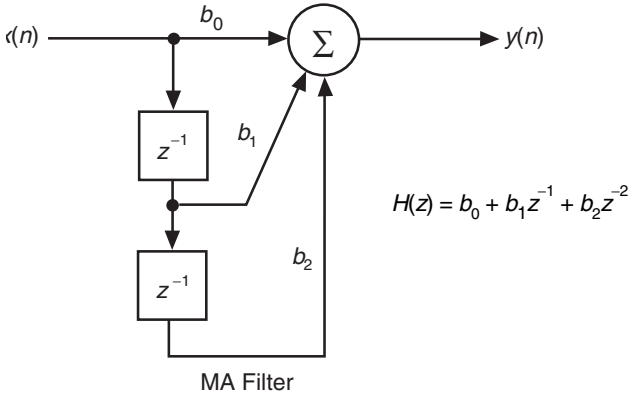
Referring again to the three filter structures in Figure 1.33, it is possible to create any arbitrary transfer function $H(z)$ with any one of the three structures. However, the orders of the realizations will be very different for one structure as compared to another. For instance, an infinite-order MA filter may be required to duplicate an M^{th} order AR filter.

One of the most basic theorems of adaptive and optimal filter theory is the *Wold decomposition*. This theorem states that any real-world process can be decomposed into a deterministic component (such as a sum of sine waves at specified amplitudes, phases, and frequencies) and a noise process. In addition, the theorem states that the noise process can be modeled as the output of a linear filter excited at its input by a white noise signal.

1.8 ADAPTIVE FILTERS AND SYSTEMS

The problem of determining the optimum linear filter was solved by Norbert Wiener and others. The solution is referred to as the *Wiener filter* and is discussed in Section 1.8.1. Adaptive filters and adaptive systems attempt to find an optimum set of filter parameters





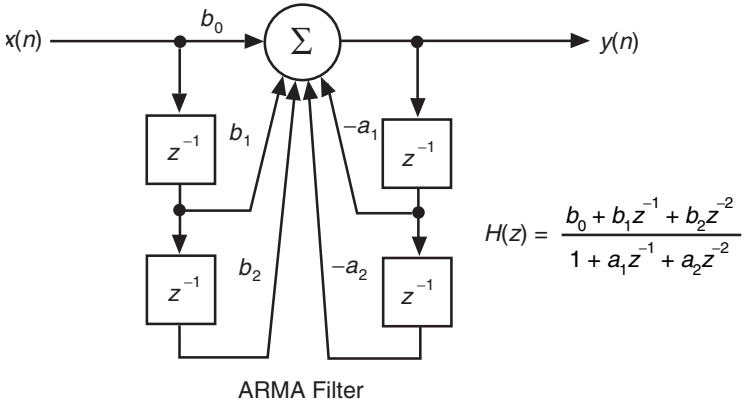


FIGURE 1.33 AR, MA, and ARMA filter structures.

(often by approximating the Wiener optimum filter) based on the time-varying input and output signals. In this section, adaptive filters and their application in closed-loop adaptive systems are discussed briefly. Closed-loop adaptive systems are distinguished from open-loop systems by the fact that in a closed loop system the adaptive processor is controlled based on information obtained from the input signal and the output signal of the processor. Figure 1.34 illustrates a basic adaptive system consisting of a processor that is controlled by an adaptive algorithm, which is in turn controlled by a performance calculation algorithm that has direct knowledge of the input and output signals.

Closed-loop adaptive systems have the advantage that the performance calculation algorithm can continuously monitor the input signal (d) and the output signal (y) and determine if the performance of the system is within acceptable limits. However, because several feedback loops may exist in this adaptive structure, the automatic optimization algorithm may be difficult to design, the system may become unstable, or the system may result in nonunique and/or nonoptimum solutions. In other situations, the adaptation process may not converge and lead to a system with grossly poor performance. In spite of these possible drawbacks, closed-loop adaptive systems are widely used in communications, digital storage systems, radar, sonar, and biomedical systems.

The general adaptive system shown in Figure 1.34(a) can be applied in several ways. The most common application is prediction where the desired signal (d) is the application provided input signal and a delayed version of the input signal is provided to the input of the adaptive processor (x) as shown in Figure 1.34(b). The adaptive processor must then try to predict the current input signal in order to reduce the error signal (ε) toward a mean squared value of zero. Prediction is often used in signal encoding (e.g., speech compression) because if the next values of a signal can be accurately predicted, then these samples need not be transmitted or stored. Prediction can also be used to reduce noise or interference and therefore enhance the signal quality if the adaptive processor is designed to only predict the signal and ignore random noise elements or known interference patterns.

As shown in Figure 1.34(c), another application of adaptive systems is system modeling of an unknown or difficult to characterize system. The desired signal (d) is the unknown system's output, and the input to the unknown system and the adaptive processor (x) is a broadband test signal (perhaps white Gaussian noise). After adaptation, the unknown system is modeled by the final transfer function of the adaptive processor. By using an AR, MA, or ARMA adaptive processor, different system models can be obtained. The magnitude of the error (ε) can be used to judge the relative success of each model.

1.8.1 Wiener Filter Theory

The problem of determining the optimum linear filter given the structure shown in Figure 1.35 was solved by Norbert Wiener and others. The solution is often referred to as the *Wiener filter.* The statement of the problem is as follows:

Determine a set of coefficients, w_k, that minimize the mean of the squared error of the filtered output as compared to some desired output. The error is written

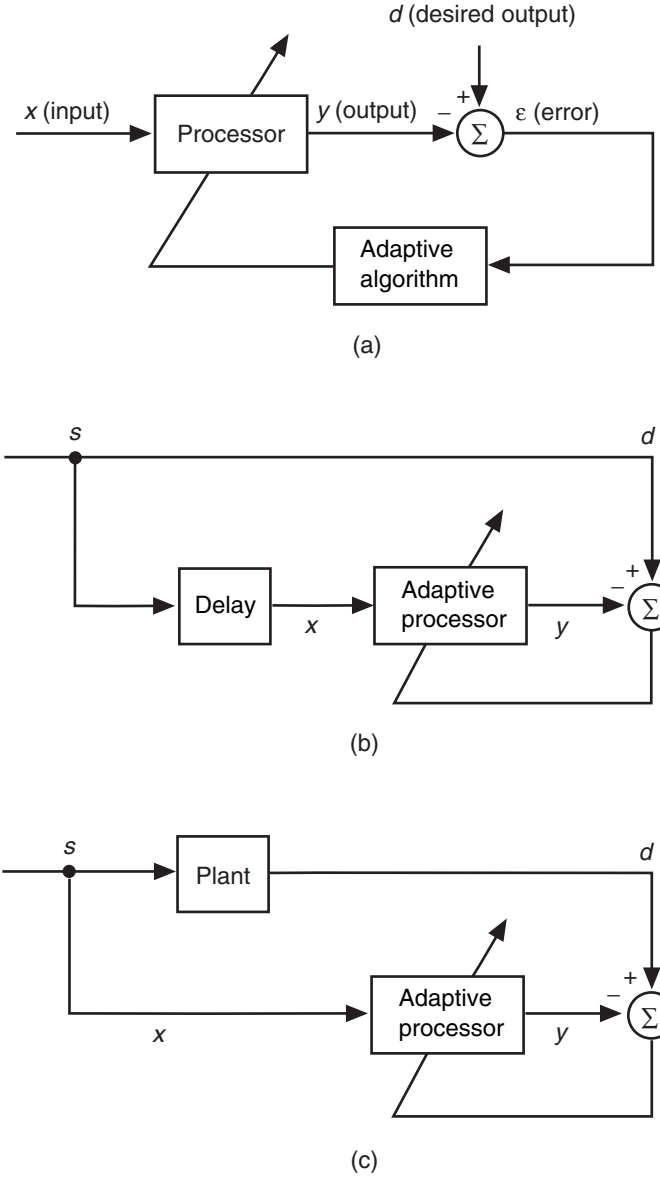


FIGURE 1.34 (a) Closed-loop adaptive system. (b) Prediction. (c) System modeling.

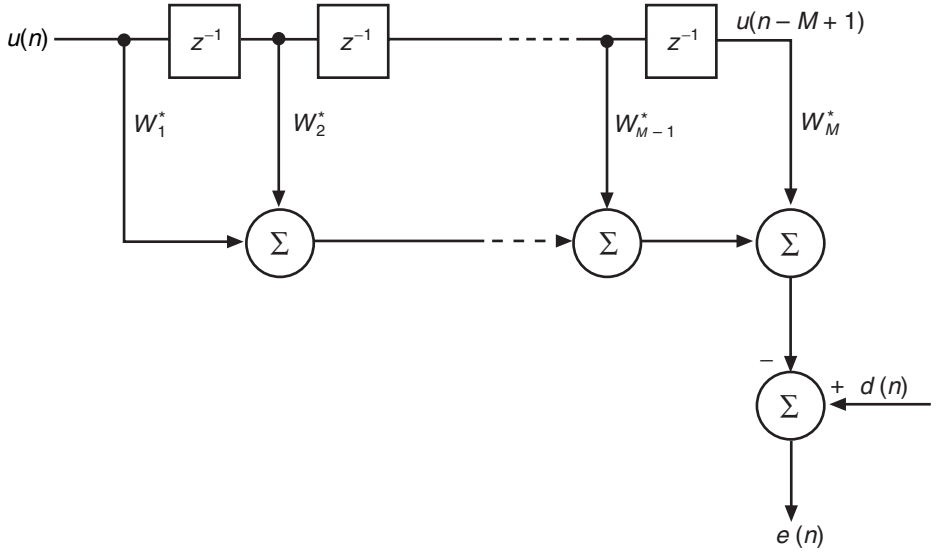


FIGURE 1.35 Wiener filter problem.

$$e(n) = d(n) - \sum_{k=1}^{M} w_k^* u(n-k+1) \tag{1.116}$$

or in vector form

$$e(n) = d(n) - \mathbf{w}^H \mathbf{u}(n) \tag{1.117}$$

The mean squared error is a function of the tap weight vector $\mathbf{w}$ chosen and is written

$$J(\mathbf{w}) = \mathrm{E}\{e(n)e^*(n)\} \tag{1.118}$$

Substituting in the expression for $e(n)$ gives

$$J(\mathbf{w}) = \mathrm{E}\{d(n)d^*(n) - d(n)\mathbf{u}^H(n)\mathbf{w} - \mathbf{w}^H\mathbf{u}(n)d^*(n) + \mathbf{w}^H\mathbf{u}(n)\mathbf{u}^H(n)\mathbf{w}\} \tag{1.119}$$

$$J(\mathbf{w}) = var[d] - \mathbf{p}^H\mathbf{w} - \mathbf{w}^H\mathbf{p} + \mathbf{w}^H\mathbf{R}\mathbf{w} \tag{1.120}$$

where $\mathbf{p} = \mathrm{E}\{\mathbf{u}(n)d^*(n)\}$, the vector that is the product of the cross correlation between the desired signal and each element of the input vector.

In order to minimize $J(\mathbf{w})$ with respect to $\mathbf{w}$, the tap weight vector, one must set the derivative of $J(\mathbf{w})$ with respect to $\mathbf{w}$ equal to zero. This will give an equation that when solved for $\mathbf{w}$ gives $\mathbf{w}_0$, the optimum value of $\mathbf{w}$.

If the complex quantity w_k is equal to $a_k + jb_k$, where a and b are real numbers, then

$$\frac{dJ(\mathbf{w})}{d\mathbf{w}} = \begin{bmatrix} \dfrac{\partial j}{\partial a_1} + \dfrac{\partial J}{\partial b_1} \\ \dfrac{\partial J}{\partial a_2} + \dfrac{\partial J}{\partial b_2} \\ \vdots \\ \dfrac{\partial J}{\partial a_M} + j\dfrac{\partial J}{\partial b_M} \end{bmatrix} \tag{1.121}$$

Taking the components of $J(\mathrm{w})$ singly

$$\frac{d(\mathrm{var}\{d\})}{d\mathbf{w}} = 0 \tag{1.122}$$

$$\frac{d(\mathbf{p}^H\mathbf{w})}{d\mathbf{w}} = 0 \tag{1.123}$$

$$\frac{d(\mathbf{w}^H\mathbf{w})}{d\mathbf{w}} = 2\mathbf{p} \tag{1.124}$$

$$\frac{d(\mathbf{w}^H\mathbf{R}\mathbf{w})}{d\mathbf{w}} = 2\mathbf{R}\mathbf{w} \tag{1.125}$$

Now setting the total derivative equal to zero

$$-2\mathbf{p} + 2\mathbf{R}\mathbf{w}_0 = 0 \tag{1.126}$$

or

$$\mathbf{R}\mathbf{w}_0 = \mathbf{p} \tag{1.127}$$

If the matrix $\mathbf{R}$ is invertible (nonsingular), then $\mathbf{w}_0$ can be solved as

$$\mathbf{w}_0 = \mathbf{R}^{-1}\mathbf{p} \tag{1.128}$$

So the optimum tap weight vector depends on the autocorrelation of the input process and the cross correlation between the input process and the desired output. Equation is called the *normal equation* because a filter derived from this equation will produce an error that is orthogonal (or normal) to each element of the input vector. This can be written

$$\mathrm{E}\{\mathrm{u}(n)e_0^*(n)\} = 0 \tag{1.129}$$

It is helpful at this point to consider what must be known to solve the Wiener filter problem:

1. M x M autocorrelation matrix of $\mathbf{u}(n)$, the input vector
2. Cross-correlation vector between $\mathbf{u}(n)$ and $d(n)$ the desired response

It is clear that knowledge of any individual $\mathbf{u}(n)$ will not be sufficient to calculate these statistics. One must take the ensemble average, E{}, to form both the autocorrelation and the cross correlation. In practice, a model is developed for the input process, and from this model the second order statistics are derived.

A legitimate question at this point is: What is $d(n)$? It depends on the problem. One example of the use of Wiener filter theory is in linear predictive filtering. In this case, the desired signal is the next value of $u(n)$, the input. The actual $u(n)$ is always available one sample after the prediction is made, and this gives the ideal check on the quality of the prediction.

1.8.2 LMS Algorithms

The LMS algorithm is the simplest and most used adaptive algorithm today. In this brief section, the LMS algorithm as it is applied to the adaptation of time-varying FIR filters (MA systems) and IIR filters (adaptive recursive filters or ARMA systems) is described. A detailed derivation, justification, and a discussion concerning convergence properties can be found in the References (see Haykin or Stearns and David).

For the adaptive FIR system the transfer function is described by

$$y(n) = \sum_{q=0}^{Q-1} b_q(k)x(n-q) \tag{1.130}$$

where $b(k)$ indicates the time-varying coefficients of the filter. With an FIR filter the mean-squared error performance surface in the multidimensional space of the filter coefficients is a quadratic function and has a single minimum mean-squared error (MMSE). The coefficient values at the optimal solution is called the MMSE solution. The goal of the adaptive process is to adjust the filter coefficients in such a way that they move from their current position toward the MMSE solution. If the input signal changes with time, the adaptive system must continually adjust the coefficients to follow the MMSE solution. In practice, the MMSE solution is often never reached.

The LMS algorithm updates the filter coefficients based on the method of steepest descent. This can be described in vector notation as follows:

$$\mathbf{B}_{k+1} = \mathbf{B}_k - \mu\nabla_k \tag{1.131}$$

where $\mathbf{B}_k$ is the coefficient column vector, μ is a parameter that controls the rate of convergence, and the gradient is approximated as

$$\nabla_k = \frac{\partial E\left[\varepsilon_k^2\right]}{\partial \mathbf{B}_k} \cong -2\varepsilon_k \mathbf{X}_k \tag{1.132}$$

where $\mathbf{X}_k$ is the input signal column vector and ε_k is the error signal as shown on Figure 1.34. Thus, the basic LMS algorithm can be written as

$$\mathbf{B}_{k+1} = \mathbf{B}_k + 2\mu\varepsilon_k\mathbf{X}_k \tag{1.133}$$

The selection of the convergence parameter must be done carefully because if it is too small the coefficient vector will adapt very slowly and may not react to changes in the input signal. If the convergence parameter is too large, the system will adapt to noise in the signal and may never converge to the MMSE solution.

For the adaptive IIR system the transfer function is described by

$$y(n) = \sum_{q=0}^{Q-1} b_q(k)x(n-q) - \sum_{p=1}^{P-1} a_p(k)y(n-p) \tag{1.134}$$

where *b(k)* and *a(k)* indicate the time-varying coefficients of the filter. With an IIR filter the mean-squared error performance surface in the multidimensional space of the filter coefficients is not a quadratic function and can have multiple minimum, which may cause the adaptive algorithm never to reach the MMSE solution. Because the IIR system has poles, the system can become unstable if the poles ever move outside the unit circle during the adaptive process. These two potential problems are serious disadvantages of adaptive recursive filters, which limits their application and complexity. For this reason, most applications are limited to a small number of poles. The LMS algorithm can again be used to update the filter coefficients based on the method of steepest descent. This can be described in vector notation as follows:

$$\mathbf{W}_{k+1} = \mathbf{W}_k - \mathbf{M}\nabla_k \tag{1.135}$$

where $\mathbf{W}_k$ is the coefficient column vector containing the *a* and *b* coefficients, **M** is a diagonal matrix containing convergence parameters μ for the *a* coefficients, and ν_0 through ν_{P-1} that controls the rate of convergence of the *b* coefficients. In this case, the gradient is approximated as

$$\nabla_k \cong -2\varepsilon_k\left[\alpha_0 \ldots \alpha_{Q-1}\beta_1 \ldots \beta_P\right]^T \tag{1.136}$$

where ε_k is the error signal as shown in Figure 1.34 and

$$\alpha_n(k) = x(k-n) + \sum_{q=0}^{Q-1} b_q(k)\alpha_n(k-q) \tag{1.137}$$

$$\beta_n(k) = y(k-n) + \sum_{p=0}^{P-1} b_q(k)\beta_n(k-p) \tag{1.138}$$

The selection of the convergence parameters must be done carefully because if they are too small the coefficient vector will adapt very slowly and may not react to changes in the input signal. If the convergence parameters are too large, the system will adapt to

noise in the signal or may become unstable. The proposed new location of the poles should also be tested before each update to determine if an unstable adaptive filter is about to be used. If an unstable pole location is found, the update should not take place and the next update value may lead to a better solution.

1.9 TWO-DIMENSIONAL SIGNAL PROCESSING

The theory presented in this chapter for one-dimensional signal processing can be extended to two dimensions quite naturally. Two-dimensional signal processing is most often used in the manipulation and enhancement of images of all types. An image is a two-dimensional representation of a group of objects or object characteristics. The sources of illumination that produced the original scene on which the image is based could be light, sound waves, or radar beams. Some techniques such as infrared imaging use energy radiated from the scene itself to produce the image. The common aspect of all forms of imaging considered here is that the scene is represented on an imaging plane by an array of numbers. In visible light images, the array of numbers correspond to light intensity levels and, in radar and sonar images, the numbers correspond to amplitudes of the returned echo.

We most often presented one-dimensional signal processing as a process performed on time sequences. In two-dimensional processing, the independent variables are usually the Cartesian coordinates of position on the image plane. Instead of sequences of numbers, we have two-dimensional arrays of numbers upon which to perform the signal-processing operations.

1.9.1 The Two-Dimensional Fourier Transform

The Fourier transform in two dimensions is written

$$F(u,v) = \frac{1}{\text{Area}} \iint\limits_{x,y\ region} f(x,y)e^{-j2\pi(ux+vy)}dxdy$$

and the inverse Fourier transform is then

$$f(x,y) = \iint\limits_{all\ u,v} F(u,v)e^{+j2\pi(ux+vy)}dudv$$

Using an analogy to the development in one dimension we can show that a valid two-dimensional discrete Fourier transform is written

$$F(u,v) = \frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} F(m,n)\exp\left[-j2\pi\left(\frac{um}{M}+\frac{vn}{N}\right)\right] \tag{1.139}$$

where u and v have become integer indices instead of continuous variables. The inverse DFT is

$$f(m,n) = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} F(u,v) \exp\left[+j2\pi \left(\frac{um}{M} + \frac{(vn)}{N} \right) \right] \tag{1.140}$$

Beware that in different texts the position of the factors of $1/M$ and $1/N$ migrate from $F(u,v)$ to $f(m,n)$. Figure 1.36(a) shows an example 128 x 128 image consisting of repeated white (pixel value = 200) and black (pixel value = 0) areas. Figure 1.36(b) is the two-dimensional FFT output. Note that the "frequency" of the x-axis direction is lower in Figure 1.36(a) but this appears in the y-axis direction of the two-dimensional FFT output. The multiple points in the y-axis direction in Figure 1.36(b) is a result of the third, fifth, and seventh harmonics in the white and black blocks in the input image. The three points in the x-axis direction in Figure 1.36(b) is a result of the higher frequency content along the y-axis of the input image (the center point is the DC level of the input image). Also, the diagonal to the right in the input image becomes a diagonal to the left in the frequency domain output.

1.9.2 Two-Dimensional Convolution

Filtering in two dimensions is usually presented in terms of the convolution equation. This equation employs a two-dimensional impulse response, $h(m,n)$, and is written

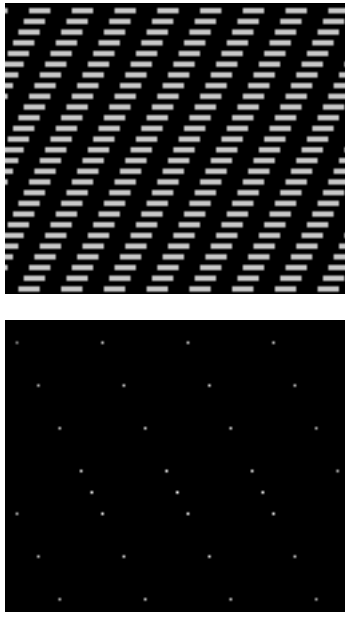


FIGURE 1.36. Two-dimensional Fourier transform example. (a) Input binary image (128 x 128). (b) Two-dimensional FFT magnitude output.

$$g(p,q) = \sum_{m=0}^{N-1} \sum_{m=0}^{N-1} h(m,n) f(p-m, q-n) \tag{1.141}$$

This equation can be visualized using the pictures in Figure 1.37. First, the input image, $f(m,n)$, is shown in Figure 1.37(a). It is defined over the region $m = 0$ to $N - 1$ and $n = 0$ to $N - 1$. Figure 1.37(b) shows $f(-m,-n)$, which is a simple rotation of $f(m,n)$ about the m and n axes. Figure 1.37(c) then shows $f(-m,-n)$ translated by p and q, which is $f(p - m, q - n)$. Now the impulse $h(m,n)$ is overlayed on the rotated, translated input image, and a point for point multiplication is performed. Each point in the output image is generated by the summation of these multiplications at a given translation of $f(-m,-n)$. One can imagine $f(-m,-n)$ sliding over the fixed $h(m,n)$, and at each new position a multiplication and summation is performed.

Any two-dimensional linear shift invariant operator can be represented by the *convolution equation.* Two examples will illustrate convolution: First, an impulse function that has zero value at all m,n except at 0,0 where it equals 1. One can see from the convolution equation that $g(p,q)$ is equal to $f(p,q)$. Therefore, the impulse function is the identity operator.

Next, we choose an impulse response that is equal to 1 for a small neighborhood around the origin

$$h(m,n) = \begin{cases} 1, & \text{for } 0 \le m < k \text{ and } 0 \le n < k \\ 0, & \text{otherwise} \end{cases}$$

This is shown in Figure 1.38. This impulse response will have the effect of averaging a neighborhood of points in the input image to produce each point in the output. The effect on an image is analogous to a moving average lowpass filter in the time domain. The higher frequency components of the image are subdued, giving blurred edges and overall softening and smoothing of textures.

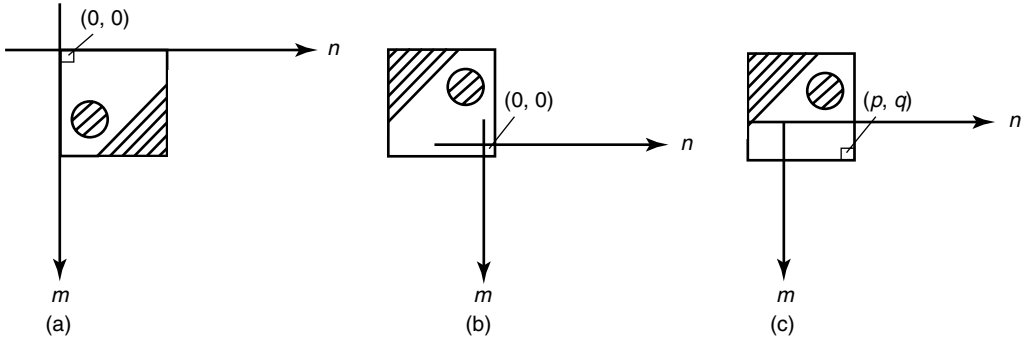


FIGURE 1.37 (a) Original convolution kernel. (b) Index reversed. (c) Index reversed and shifted.

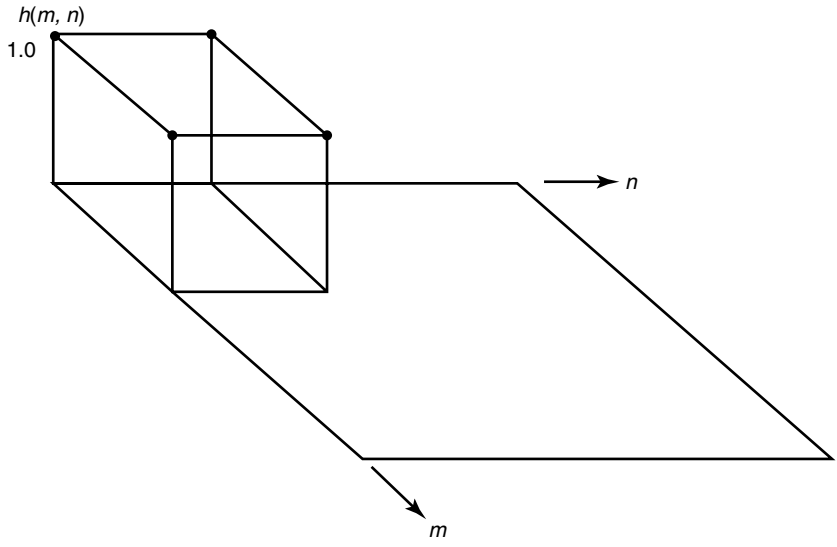


FIGURE 1.38 Simple impulse response.

This is a good point at which to mention edge effects in convolution. Picture the results of translating the input image through its full range when the impulse is as shown in Figure 1.38. This example is typical of many image-processing computations in which the impulse response is small in extent compared to the input and output image. For the majority of points in the output image $h(m,n)$ completely overlaps $f(p - m, q - n)$. There are no points in the impulse response without an input image point to multiply. However, when the translation is such that the overlap of the two square arrays is similar to Figure 1.39 there will be output image points generated without a full set of input image points to draw on.

The usual way to handle this dilemma is to use zero values for the input image at these points. A full-size output image can then be generated. In the example of the averaging operator, the edge points of the output image would tend to decrease in intensity due to the larger number of zeroes in the input. One could also use an average of the complete set of input values rather than simply zero these points. However, these edge points are always somewhat erroneous when compared with values at the center of the output image, and in some applications they are simply discarded.

1.9.3 Using the FFT to Speed Up Two-Dimensional Processing

As with the one-dimensional DFT, the two-dimensional DFT described in Section 1.8.1 would not be very practical if there were not a method to decrease the number of computations required in its calculation. There is such a method based on the one-dimensional FFT. First, the 2D DFT is separated into two sums

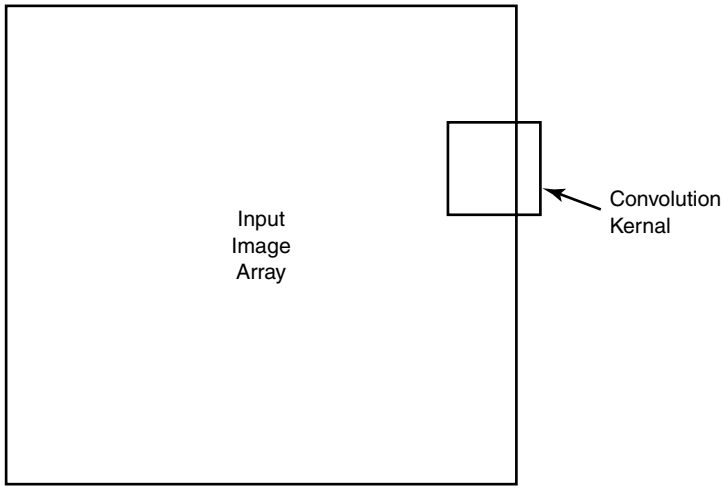


FIGURE 1.39 Image convolution showing edge effects.

$$F(u,v) = \frac{1}{MN}\sum_{m=0}^{M-1}\exp\left[\frac{-j2\pi um}{M}\right]\left\{\sum_{n=0}^{N-1} f(m,n)\exp\left[\frac{-j2\pi vn}{N}\right]\right\} \tag{1.142}$$

If the value of m is considered a parameter, the summation inside the brackets, { }, in this equation has the form of a one-dimensional DFT. This means that an FFT can be used to perform this computation for each value of m, a total of M FFTs in all. When completed these computations will produce a matrix of values, say, $g(m,v)$, whose indices are m and v. For each value of v there is a function of m that could be called $g_v(m)$. The equation can now be written

$$F(u,v) = \frac{1}{MN}\sum_{m=0}^{M-1} g_v(m)\exp\left[\frac{-j2\pi um}{M}\right] \tag{1.143}$$

This set of computations also has the form of a DFT and can be computed using a separate FFT for each value of v, this time a total of N FFTs. This technique can be used equally well for the inverse two-dimensional DFT.

1.9.4 Two-Dimensional Filtering in the Transform Domain

One method of filtering a two-dimensional signal uses the FFT for fast convolution. This involves an extension of the *convolution theorem* to two dimensions. Taking the DFT of both sides of the convolution equation gives

$$G(u,v) = H(u,v)F(u,v) \tag{1.144}$$

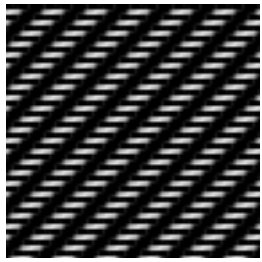


FIGURE 1.40 Image after frequency domain filtering by setting the upper frequencies to zero.

where each frequency domain representation has been derived from the formula for the two-dimensional DFT.

It is common for the extent of the impulse response $h(m,n)$ to be smaller in the (m,n) plane than the input image. The question then becomes what size DFT should be used to perform each transformation. For the convolution theorem to be valid, the same-size DFT must be used for each transformation and it should naturally be at least as big as the larger of h and f. Also, to avoid wraparound effects due to the periodicity of the DFT, it is necessary to pad the input image $f(m,n)$ with a number of zeroes in each dimension equal to the size of the impulse response. For instance, if the filter kernel (or impulse response) is an M x M array and the input image $f(m,n)$ is an N x N array (where N is usually greater than M), each array would be increased in size to $(M + N)$ x $(M + N)$ by filling with zeroes. The DFT would then be performed on each array.

An additional consideration arises when using the FFT to perform the two-dimensional DFTs required for convolution. The arrays must be a power of 2 (for a radix 2 FFT) in length on each side. This may require some extra zero filling. After $H(u,v)$ and $F(u,v)$ are obtained, they are multiplied point by point to create the output array $G(u,v)$. An *inverse two-dimensional FFT* is then performed on this array to obtain the final resultant image $g(m,n)$. Figure 1.40 shows the output of frequency domain filtering of the image shown in Figure 1.36(a). The high-frequency components in the image were removed by setting the upper three-quarters of FFT result to zero. Note that the image lacks edge definition, but the blocks are still well defined even with only one-quarter of the frequency domain data. For the general filter case, a total of three two-dimensional FFT operations must be performed (the FFT of the filter kernal can be stored and used repeatedly with other images). Depending on the relative sizes of the image and filter kernel, the two-dimensional FFT method may not be more computationally efficient. Further examples of two-dimensional convolution can be found in Chapter 7, where image-processing functions are considered in more detail.

1.10 REFERENCES

BRIGHAM, E., *The Fast Fourier Transform,* Prentice Hall, Englewood Cliffs, NJ, 1974.

CLARKSON, P., *Optimal and Adaptive Signal Processing,* CRC Press, Boca Raton, FL, 1993.

ELLIOTT, D., ed., *Handbook of Digital Signal Processing,* Academic Press, San Diego, 1987.

EMBREE, P., *C Algorithms for Real-Time DSP,* Prentice Hall, Englewood Cliffs, NJ, 1995.

EMBREE, P., and KIMBLE, B., *C Language Algorithms for Digital Signal Processing,* Prentice Hall, Englewood Cliffs, NJ, 1991.

HARRIS, F., "On the Use of Windows for Harmonic Analysis With the Discrete Fourier Transform," *Proceedings of the IEEE,* Vol. 66, No. 1, pp. 51–83, 1978.

HAYKIN, S., *Adaptive Filter Theory,* Prentice Hall, Englewood Cliffs, NJ, 1986.

MCCLELLAN, J., PARKS, T., and RABINER, L., "A Computer Program for Designing Optimum FIR Linear Phase Digital Filters," *IEEE Transactions on Audio and Electro-acoustics,* AU-21, No. 6, pp. 506–526, 1973.

MOLER, C., LITTLE, J., and BANGERT, S., *PC-MATLAB User's Guide,* Math Works, Sherborn, MA, 1987.

OPPENHEIM, A., and SCHAFER, R., *Digital Signal Processing,* Prentice Hall, Englewood Cliffs, NJ, 1975.

OPPENHEIM, A., and SCHAFER, R., *Discrete-Time Signal Processing,* Prentice Hall, Englewood Cliffs, NJ, 1989.

PAPOULIS, A., *Probability, Random Variables and Stochastic Processes,* McGraw-Hill, New York, 1965.

RABINER, L., and GOLD, B., *Theory and Application of Digital Signal Processing,* Prentice Hall, Englewood Cliffs, NJ, 1975.

STEARNS, S., and DAVID, R., *Signal Processing Algorithms,* Prentice Hall, Englewood Cliffs, NJ, 1988.

STEARNS, S., and DAVID, R., *Signal Processing Algorithms in FORTRAN and C,* Prentice Hall, Englewood Cliffs, NJ, 1993.

VAIDYANATHAN, P., *Multirate Systems and Filter Banks,* Prentice Hall, Englewood Cliffs, NJ, 1993.

WIDROW, B., and STEARNS, S., *Adaptive Signal Processing,* Prentice Hall, Englewood Cliffs, NJ, 1985.

CHAPTER 2

Programming Fundamentals

The purpose of this chapter is to provide the programmer with a complete overview of the fundamentals of the C and C++ programming languages that are important in DSP applications. In particular, bitfields, enumerated data types, and unions are not discussed, because they have limited utility in the majority of DSP programs. Readers with prior programming experience may wish to skip the bulk of this chapter with the possible exception of the more advanced concepts related to pointers, structures, and classes presented in Sections 2.7, 2.8, and 2.9. The proper use of pointers and data structures can make a DSP program easier to write and much easier for others to understand. Programmers with C experience who are new to C++ should pay particular attention to sections describing classes and operator overloading. Example DSP programs in this and the following chapters will clarify the importance of pointers and data structures in DSP programs.

2.1 THE ELEMENTS OF DSP PROGRAMMING

The purpose of a *programming language* is to provide a tool so that a programmer can easily solve a problem involving the manipulation of some type of information. Based on this definition, the purpose of a DSP program is to manipulate a signal (a special kind of information) in such a way that the program solves a signal-processing problem. To do this, a DSP programming language must be able to

1. Organize different types of data (variables and data types)
2. Describe the operations to be done (operators)

3. Control the operations that are performed based on the results of operations (program control)
4. Organize the data and the operations so that a sequence of program steps can be executed from anywhere in the program (functions and data structures)
5. Move data back and forth between the outside world and the program (input/output)

These five elements are required for efficient programming of DSP algorithms. Their implementation is described in the remainder of this chapter.

As a preview of the C++ programming language, a simple DSP program is shown in Listing 2.1. It illustrates each of the five elements of DSP programming. The listing is divided into sections as indicated by the comments in the program. This simple DSP program reads a series of numbers from the keyboard and determines the average and variance of the numbers that were entered. In signal-processing terms, the signal is a series of numbers entered by the user and the output is the DC level and total AC power of the signal. A typical dialogue when using this program is as follows:

```
Number of input values ? 6

Enter values:
2.54
1.54
1.00
1.46                              user inputs six signal values
2.46
3.00

Input Signal Entered was as follows:

Index    Value
 0     2.54000
 1     1.54000
 2     1.00000
 3     1.46000                    program displays signal values
 4     2.46000
 5     3.00000

Average = 2.00000
Variance = 0.601280               program displays results
Standard Deviation = 0.775422
```

Section 1 of Listing 2.1 includes header files for the standard input/output and math libraries that are discussed in Appendix A. Listing 2.1 is shown in a format intended to make it easier to follow and modify. The beginning of each section of the program is indicated by single-line comments in the *program source code* (i.e., **// section 4**). A single-line comment ends at the end of the line. Multiline comments can be specified between the **/*** and ***/** delimiters.

```
#include <iostream.h>                          // section 1
#include <math.h>

float average( const float *array, int size ) // section 2
{
    if( array == NULL || size == 0 )          // validate parameters
        throw "Invalid parameters";
    float sum = array[0];                     // initialize and sum
    for( int i = 1; i < size; i++ )
        sum = sum + array[i];                 // calculate sum
    return( sum / size );                     // return average
}

float variance( const float *array, int size ) // section 3
{
    if( array == NULL || size < 2 )           // validate parameters
        throw "Invalid parameters";

    float sum = array[0];                     // initialize signal sum
    float sum2 = array[0] * array[0];         // sum of signal squared
    for( int i = 1; i < size; i++ )
    {
        sum = sum + array[i];
        sum2 = sum2 + array[i] * array[i];    // calculate both sums
    }
    float ave = sum / size;                   // calculate average
    return((sum2 - sum * ave)/( size - 1)); // return variance
}

void main()                                   // section 4
{
    int number = 0;                           // section 5
    do
    {
        cout << "Number of input values ? ";
        cin >> number;                        // number of samples
        if( cin.eof() )
            return;
    }
    while( number <= 0 );                     // repeat until valid

    float *signal = new float[number];        // section 6
    if( signal == NULL )                      // check allocation
```

LISTING 2.1 Example C++ program that calculates the average and variance of a sequence of numbers entered by the user. (*Continued*)

```
    {
       cerr << "Cannot allocate memory\n"; // report error and exit
        return;
    }
    cout << "\nEnter values:\n";            // section 7
    for( int count = 0; count < number; count++ )
    {
        cin >> signal[count];               // read input signal
        if( cin.eof() )
            break;
    }

    cout.flags( ios::showpoint );           // section 8
    cout << "\nInput Signal Entered was as follows:\n";
    cout << "\nIndex    Value\n";
    for( int i = 0; i < count; i++ )
        cout << " " << i << "     " << signal[i] << endl;

    try                                     // section 9
    {
        float ave = average( signal, count );
        float var = variance( signal, count );
        cout << "\n\nAverage = " << ave;    // print results
        cout << "\nVariance = " << var;
        cout << "\nStandard Deviation = " << sqrt( var ) << endl;
    }
    catch( const char *e )
    {
        cerr << e << endl;                  // section 10
    }
    delete [] signal;                      // section 11
}
```

LISTING 2.1 (*Continued*)

Sections 2 and 3 represent two functions used by the program to calculate the average and variance, respectively. Since the **array** parameter is declared as **const float** *, the compiler will ensure that nothing can be assigned into the memory pointed to by **array**. In each function, the input parameters are checked and if there are NULL-pointer or divide-by-zero errors, an exception is thrown that is immediately handled by the program in section 10. These functions relate primarily to element two (operators), since the detailed operation of each function is defined. Each function is defined by the first line in sections 2 and 3, which indicates the type of the return value and the types of arguments. The function arguments are checked, and the local variables to be used in each function

are declared. The operations required by each function are then defined followed by a **return** statement, which passes the result back to the main program.

Section 4 declares the **main**() routine, which is executed first and is responsible for the control of the program. The main routine is declared as **void**, which tells the compiler that it doesn't return any values. As shown in this example, all statements in C and C++ end in a semicolon (**;**) and may be placed anywhere on the input line (in fact, all spaces and carriage control characters outside of quotes are ignored by compilers). This section of Listing 2.1 relates to program organization (element four of the above list).

Some variables are declared as *single floating-point numbers* (such as ave and var), some variables are declared as *single integers* (such as **i**, **count**, and **number**), and some variables are *arrays* (such as **signal**). This program section relates to element one, data organization. These variables are declared close to where they are used so that it is clear what type of variable they are when looking at the code.

Sections 5 and 7 declare variables to be used for the number of samples in the input signal, prompt the user to enter the data, and receive the input from the user. They check if the user ended the input prematurely. The **cin** and **cout** classes represent standard screen input and output, and they are accessed by the overloaded **>>** and **<<** operators described in Section 2.3.5. Section 7 reads the input into the array using a **for** *loop* (similar to a DO loop in FORTRAN). If the user enters an incorrect floating-point value, the input sequence is terminated and control is passed to the next part of the program, which displays the correctly entered values as shown in the example above. This section relates to element five, input/output, and element three, program control of our list.

Section 6 declares a variable that contains the location of memory allocated for the number of floating-point values specified by the user. If the memory allocation fails, the user is alerted with the standard error output class **cerr**, and the program exits (see **iostream.h**).

Section 8 formats the output to show the floating-point values and then prints out the values in a tabular format. The **endl** at the end of the **cout** statement is a constant that moves the output to the next line. The six numbers entered in the above dialogue represent six samples of a single cycle of an offset cosine function with an AC peak amplitude of one and a DC offset of two. The output values shown in the dialogue indicate that the average value of the six input samples is 2.0, the variance (or power) of the cosine samples is 0.601, and the standard deviation is 0.775.

Since the **average** and **variance** functions can raise an exception (for NULL-pointers or divide by zero), they are placed inside of the try-block in section 9 so that if an error occurs, program control is immediately moved to the exception handler specified by the **catch** keyword. Exception handling may seem like overkill for this simple example, but imagine larger programs where errors can occur anywhere. Instead of checking return values from every function and calling the appropriate error handler explicitly, exception handling in C++ can greatly increase the ease of writing reliable code (see Section 2.4.6). This part of the program relates to element four of our list (functions and data structures) because the operations defined in functions **average** and **variance** are executed and stored. After the average and variance have been calculated, the results are printed and control flow moves past the exception handler to section 11.

The exception handler is listed in section 10. Since the exception thrown was a string of characters (or **char ***), this handler will be called when an error occurs. The exception handler displays the exception on the standard error output and continues to section 11.

The program ends in section 11 after the memory allocated in section 6 is released with the **delete** command. The pair of brackets between the delete command and the variable tell the compiler that the variable was an array.

The rest of this chapter will provide much more detailed information concerning the basic elements of the C++ programming language that are well suited for DSP programs.

2.2 VARIABLES AND DATA TYPES

All programs work by manipulating some kind of information. A variable is defined by declaring that a sequence of characters (the variable identifier or name) are to be treated as a particular predefined type of data. An identifier may be any sequence of characters (usually with some length restrictions) that obeys the following three rules:

1. All identifiers start with a letter or an underscore (_).
2. The rest of the identifier can consist of letters, underscores, and/or digits.
3. The rest of the identifier does not match any of the language keywords (check compiler implementation for a list of these).

In particular, the name of the variable is case sensitive, making **Average**, **AVERAGE**, and **AVeRaGe** all different.

The C and C++ languages support several different data types that represent integers (declared int), floating-point numbers (declared **float** or **double**), and text data (declared **char**). Also, arrays of each variable type and pointers of each type may be declared. The first two types of numbers will be covered first, followed by a brief introduction to arrays (they are covered in more detail with pointers in Section 2.7). The special treatment of text using character arrays and strings will be discussed in Section 2.2.3.

2.2.1 Types of Numbers

A program must declare the variable before it is used. There are several types of numbers used depending on the format in which the numbers are stored (floating-point format or integer format) and the accuracy of the numbers (single-precision versus double-precision floating point, for example). The following example program illustrates the use of five different types of numbers:

```
#include <stdio.h>

void main()
{
```

```
    int i;              // size dependent on implementation
    short j;            // 16 bit integer
    long k;             // 32 bit integer
    float a;            // single precision floating point
    double b;           // double precision floating point
    k = 72000L;
    j = k;
    i = k;
    b = 0.1;
    a = 0.1f;

    // illustration of formatting
    printf( "\n%ld %d %d\n%20.15f\n%20.15f", k, j, i, b, a );
}
```

Three types of integer numbers (**int**, **short int**, and **long int**) and two types of floating-point numbers (**float** and **double**) are illustrated in this example. Note that the intialization of **k** (a **long int**) uses the *type suffix* **L** after the number and that the float intialization of **a** uses the type suffix **f** after the **0.1**. This strict definition of the constants avoids compiler warning messages due to conversion of the default type (**double** for floating-point constants or possibly **short** for integers) to a smaller, less acurate type. The actual sizes (in terms of the number of bytes used to store the variable) of these five types depend upon the implementation; all that is guaranteed is that a **short int** variable will not be larger than a **long int** and a **double** will be twice as large as a **float**. The size of a variable declared as just **int** depends on the compiler implementation. It is normally the size most conveniently manipulated by the target computer, thereby making programs using ints the most efficient on a particular machine. However, if the size of the integer representation is important in a program (as it often is), then declaring variables as **int** could make the program behave differently on different machines. For example, on a 16-bit machine, the above program would produce the following results:

```
72000 6464 6464
0.100000000000000
0.100000001490116
```

But on a 32-bit machine (using 32-bit **int**s), the output would be as follows:

```
72000 6464 72000
0.100000000000000
0.100000001490116
```

Note that in both cases the **short** and **long** variables, **k** and **j** (the first two numbers displayed), are the same, while the third number, indicating the **int i**, differs. In both cases, the value 6464 is obtained by masking the lower 16 bits of the 32-bit **k** value. Also, in both cases the floating-point representation of 0.1 with 32 bits (**float**) is accurate to eight decimal places (seven places is typical). With 64 bits it is accurate to at least 15 places.

Thus, to make a program truly portable, the program should contain only **short int** and **long int** declarations (these may be abbreviated **short** and **long**). In addition to the five types illustrated above, the three **ints** can be declared as unsigned by preceding the declaration with unsigned. Also, as will be discussed in more detail in Section 2.2.3 concerning text data, a variable may be declared to be only 1 byte long by declaring it a **char** (**signed** or **unsigned**). Table 2.1 gives the typical sizes and ranges of the different variable types for a 32-bit machine and a 16-bit machine (such as DOS on the IBM PC).

2.2.2 Arrays

Almost all high-level languages allow the definition of indexed lists of a given data type, commonly referred to as *arrays.* All data types can be declared as an array simply by placing the number of elements to be assigned to the array in brackets after the array name. *Multidimensional arrays* can be defined simply by appending more brackets containing the array size in each dimension. Any N-dimensional array is defined as follows:

```
type name[size1][size2] ... [sizeN];
```

For example, each of the following statements are valid array definitions:

```
unsigned int list[10];
double input[5];
short int x[2000];
char input_buffer[20];
unsigned char image[256][256];
int matrix[4][3][2];
```

TABLE 2.1 Size of and Use of Variables on Two Machines

Variable Declaration	16-bit Machine Size (bits)	16-bit Machine Range	32-bit Machine Size (bits)	32-bit Machine Range
char	8	−128 to 127	8	−128 to 127
unsigned char	8	0 to 255	8	0 to 255
int	16	−32768 to 32767	32	±2.1e9
unsigned int	16	0 to 65535	32	0 to 4.3e9
short	16	−32768 to 32767	16	−32768 to 32767
unsigned short	16	0 to 65535	16	0 to 65535
long	32	±2.1e9	32	±2.1e9
unsigned long	32	0 to 4.3e9	32	0 to 4.3e9
float	32	±1.0e±38	32	±1e±38
double	64	±1.0e±306	64	±1e±308

Note that the array definition **unsigned char image [256][256]** could define an 8-bit, 256 × 256 image plane where a gray-scale image is represented by values from 0 to 255. The last definition defines a three-dimensional matrix in a similar fashion. Arrays are referenced using brackets to enclose each index. Thus, the image array, as defined above, would be referenced as **image[i][j]**, where **i** and **j** are row and column indices, respectively. Also, the first element in all array indices is zero and the last element is *N-1*, where *N* is the size of the array in a particular dimension. Thus, an assignment of the first element of the five-element, one-dimensional array **input** (as defined above) such as **input[0]=1.3;** is legal while the result of writing to an out-of-bounds array element as in **input[5]=1.3;** is undefined. Unlike some languages, the C and C++ languages do not check the boundary conditions of array indices; it is up to the programmer to write safe code.

Arrays may be *initialized* when they are declared. The values to initialize the array are enclosed in one or more sets of braces ({}) and the values are separated by commas. For example, a one-dimensional array called vector can be declared and initialized as follows:

```
int vector[6] = { 1, 2, 3, 5, 8, 13 };
```

A two-dimensional array of six double-precision floating-point numbers can be declared and initialized using the following statement:

```
double a[3][2] =
{
    { 1.5, 2.5 },
    { 1.1e-5 , 1.7e5 },
    { 1.765 , 12.678 }
};
```

Note that commas separate the three sets of inner braces, which designate each of the three rows of the matrix a and that each array initialization is a statement that must end in a semicolon.

2.2.3 Text Data Types: Characters and Strings

Text is manipulated using arrays of 8-bit numbers declared **char**. As indicated in Table 2.1, *char variables* can be used to store numbers (a variable may be declared as **unsigned char** that can store numbers from 0 to 255) as well as characters. A single character is defined by placing single quotes around it (i.e., **'Z'** defines the 8-bit constant for Z).

Strings of characters are defined with double quotes such as **"This is a string"**. When the compiler finds a statement containing the string "This is a string" it assigns a memory area that is 17 bytes long and initializes a byte to each of the 16 characters in the string followed by a null character (ASCII code 0). The compiler then uses the starting address of the string in the statement which contained it. For example, the statement **cout << "This is a string"**) allocates the 17 bytes and passes the start address of the string (the address of the character T) to the **cout** class for display. The **cout** class will send the

string to the display device one character at a time until the null character is encountered at the end of the string. Thus, strings are said to be *null-terminated.*

Strings cannot contain nonprintable characters such as carriage returns, linefeeds, tabs, or the character to ring the "bell" (or "beep" as the case may be). In order to store these special characters in the string memory area, several special character sequences, all of which start with a backslash (\) may be used as follows:

****	defines a single backslash (\).
\'	defines an apostrophe.
\"	defines a quote character.
\n	defines a newline (carriage return and linefeed).
\t	defines a horizontal tab.
\b	defines a backspace.
\r	defines a carriage return character.
\f	defines a formfeed or clear screen.
\000	defines a character represented by ASCII code **000**, where **000** is 1 to 3 octal digits (0–7).
\xHHH	defines a character represented by ASCII code **HHH**, where **HHH** is 1 to 3 hexadecimal digits (0–9,A–F).

2.3 OPERATORS

Once variables are defined to be a given size and type, some sort of manipulation must be performed using the variables. This is done by using *operators.* The C and C++ languages have more operators than most languages; in addition to the usual assignment and arithmetic operators, they also have bitwise operators and a full set of logical operators. Some of these operators (such as bitwise operators) are especially important in order to write DSP programs that utilize the target processor efficiently.

2.3.1 Assignment Operators

The most basic operator is the *assignment operator,* which is the single equal sign (**=**). The value on the right of the equal sign is assigned to the variable on the left. Assignment statements can also be stacked as in the statement **a = b = 1;**. In this case, the statement is evaluated right to left so that 1 is assigned to **b** and **b** is assigned to **a**. In C and C++, **a = ave(x)** is an expression while **a = ave(x);** is a statement. The addition of the semicolon tells the compiler that this is all that will be done with the result from the function **ave(x)**. An expression always has a value that can be used in other expressions. Thus, **a = b + (c = ave(x));** is a legal statement. The result of this statement would be that the result returned by **ave(x)** is assigned to **c** and **b + c** is assigned to a. Multiple expressions can be placed within one statement by separating them with commas. Each expression is evalu-

ated left to right, and the entire expression (comprised of more than one expression) assumes the value of the last expression that is evaluated. For example, **a = (olda = a , ave(x));** assigns the current value of **a** to **olda**, calls the function **ave(x)**, and then assigns the value returned by **ave(x)** to **a**.

2.3.2 Arithmetic and Bitwise Operators

The usual set of binary arithmetic operators (operators that perform arithmetic on two operands) is supported using the following symbols:

*	multiplication
/	division
+	addition
–	subtraction
%	modulus (integer remainder after division)

The first four operators listed are defined for all types of variables (**char**, **int**, **float**, and **double**). The modulus operator is defined only for integer operands. Also, there is no exponent operator; this floating-point operation is supported using a *simple function call* (see Appendix A for a description of the **pow** function).

There are three unary arithmetic operators that require only one operand. First is the unary minus operator (e.g., **-i** where **i** is an **int**), which performs a two's complement change of sign of the integer operand. The unary minus is often useful when the exact hardware implementation of a digital signal processing algorithm must be simulated. The other two unary arithmetic operators are increment and decrement, represented by the symbols **++** and **−−**, respectively. These operators add or subtract one from any integer variable or pointer. The operand is often used in the middle of an expression, and the increment or decrement can be done before or after the variable is used in the expression (depending on whether the operator is before or after the variable). Although the use of **++** and **−−** is often associated with pointers (see Section 2.7), the following example illustrates these two powerful operators with the ints **i**, **j**, and **k:**

```
i = 4;
j = 7;
k = i++ + j;     /* i is incremented to 5, k = 11 */
k = k + −−j;     /* j is decremented to 6, k = 17 */
k = k + i++;     /* i is incremented to 6, k = 22 */
```

Binary bitwise operations are performed on integer operands using the following symbols:

&	bitwise AND
|	bitwise OR

^	bitwise exclusive OR
<<	arithmetic shift left (number of bits is operand)
>>	arithmetic shift right (number of bits is operand)

The unary bitwise NOT operator, which inverts all the bits in the operand is implemented with the ~ symbol. For example, if **i** is declared as an **unsigned int,** then **i = ~0;** sets i to the maximum integer value for an **unsigned int.** The bitwise NOT operator ~ should not be confused with the logical NOT operator, which is implemented with the **!** symbol. The logical NOT operator is discussed in Section 2.3.4.

2.3.3 Combined Operators

Operators can be combined with the assignment operator (=) so that almost any statement of the form

<variable> = <variable> <operator> <expression>;

can be replaced with

<variable> <operator> = <expression>;

where **<variable>** represents the same variable name in all cases. For example, the following pairs of expressions involving **x** and **y** perform the same function:

```
x = x + y;          x += y;
x = x - y;          x -= y;
x = x * y;          x *= y;
x = x / y;          x /= y;
x = x % y;          x %= y;
x = x & y;          x &= y;
x = x ^ y;          x ^= y;
x = x << y;         x <<= y;
x = x >> y;         x >>= y;
```

In many cases, the left-hand column of statements will result in a more readable and easier to understand program. For this reason, use of combined operators is often avoided. Unfortunately, some compiler implementations may generate more efficient code if the combined operator is used.

2.3.4 Logical Operators

Like all expressions, an expression involving a logical operator also has a value. A logical operator is any operator that gives a result of true or false. This could be a comparison between two values or the result of a series of ANDs and ORs. If the result of a logical operation is true, it has a nonzero value; if it is false, it has the value 0. Loops and if state-

ments (covered in Section 2.4) check the result of logical operations and change program flow accordingly. The nine logical operators are as follows:

<	less than
<=	less than or equal to
==	equal to
>=	greater than or equal to
>	greater than
!=	not equal to
&&	logical AND
||	logical OR
!	logical NOT (unary operator)

Note that **==** can easily be confused with the assignment operator (**=**) and will result in a valid expression because the assignment also has a value that is then interpreted as true or false. Also, **&&** and **||** should not be confused with their bitwise counterparts (**&** and **|**) as this may result in hard-to-find logic problems because the bitwise results may not give true or false when expected.

2.3.5 Operator Overloading

We have already witnessed operator overloading in Listing 2.1. The standard screen input and output classes have overloaded the **>>** and **<<** operators to take variables passed on the right-hand side and either load them with the values typed by the user, or display them on the screen. The **Matrix** and **Vector** classes defined later also overload these operators so that printing matrices and vectors are as easy as printing any of the predefined types in the language.

Operator overloading also covers casting from one type to another. A class can overload the **type()** operator to return a reasonable value of that type. The **String** class can return a **const char *** when a character string is needed, for example, when printing. The string class also defines the **+=** operator for concatenation, but does not define a **−=** operator because it wouldn't be clear to a user what subtracting a string would mean.

A common overloaded operator in C++ is the assignment operator =. Classes overload the assignment operator when they must perform special functions when one class is assigned to another. If a class had allocated memory for internal variables, for example, that class may want to allocate a copy of the data instead of sharing it with the assigned class. Overloading the assignment operator is a standard way of allowing the designer control over assignment behavior.

Operator overloading can simplify programming, but it must be used sparingly and only in cases where it doesn't add confusion or ambiguity about the code. A good example of ambiguity is vector multiplication. If the designer of the **Vector** class overloaded the * operator to work on two vectors, it would be unclear if the result is the scalar dot product of the two classes or the vector cross product.

2.3.6 Operator Precedence and Type Conversion

Like all computer languages, C and C++ have an operator precedence that defines which operators in an expression are evaluated first. If this order is not desired, then parentheses can be used to change the order. Thus, expressions in parenthes are evaluated first and items of equal precedence are evaluated from left to right. The operators contained in the parentheses or expression are evaluated in the following order (listed by decreasing precedence):

++,−−	increment, decrement
−	unary minus
***, /,%**	multiplication, division, modulus
+,−	addition, subtraction
<<, >>	shift left, shift right
<, <=, >=, >	relational with less than or greater than
==, !=	equal, not equal
&	bitwise AND
^	bitwise exclusive OR
|	bitwise OR
&&	logical AND
||	logical OR

Statements and expressions using the operators just described should normally use variables and constants of the same type. If, however, you mix types, C and C++ do not stop dead (like Pascal) or produce a strange unexpected result (like FORTRAN). Instead, they use a set of rules to make type conversions automatically. The two basic rules are

1. If an operation involves two types, the value with a lower "rank" is converted to the type of higher "rank." This process is called *promotion,* and the ranking from highest to lowest type is **double**, **float**, **long**, **int**, **short**, and **char**. Unsigned of each of the types outranks the individual signed type.
2. In an assignment statement, the final result is converted to the type of the variable that is being assigned. This may result in promotion or demotion where the value is truncated to a lower-ranking type.

Usually these rules work quite well, but sometimes the conversions must be stated explicitly in order to demand that a conversion be done in a certain way. This is accomplished by *type casting* the quantity by placing the name of the desired type in parentheses before the variable or expression. Thus, if **i** is an **int**, then the statement **i = 10*(1.55 + 1.67);** would set **i** to 32 (the truncation of 32.2) while the statement **i = 10*((int)1.55 + 1.67);** would set **i** to 26 [the truncation of 26.7, since **(int)1.55** is truncated to 1].

2.4 PROGRAM CONTROL

The large set of operators allows a great deal of programming flexibility for DSP applications. Programs that must perform fast binary or logical operations can do so without using special functions to do the bitwise operations. There is also a complete set of program control features that allow conditional execution or repetition of statements based on the result of an expression. Proper use of these control structures is discussed in Section 2.12.2, where structured programming techniques are considered.

2.4.1 Conditional Execution: `if-else`

The **if** statement is used to conditionally execute a series of statements based on the result of an expression. The **if** statement has the following generic format:

```
if( value )
    statement1;
else
    statement2;
```

where **value** is any expression that results in (or can be converted to) an integer value. If value is nonzero (indicating a true result), then **statement1** is executed; otherwise, **statement2** is executed. Note that the result of an expression used for **value** need not be the result of a logical operation; all that is required is that the expression results in a zero value when **statement2** should be executed instead of **statement1**. Also, the **else statement2;** portion of the above form is optional, allowing **statement1** to be skipped if value is false.

When more than one statement needs to be executed if a particular value is true, a compound statement is used. A *compound statement* consists of a left brace ({), some number of statements (each ending with a semicolon), and a right brace (}). Note that the body of the **main()** program and functions in Listing 2.1 compound statements. In fact, a single statement can be replaced by a compound statement in any of the control structures described in this section. By using compound statements, the **if-else** control structure can be nested as in the following example, which converts a floating-point number **(result)** to a two-bit two's-complement number **(out):**

```
if( result > 0 )
{
    // positive outputs
    if( result > sigma )
        out = 1;        // biggest output
    else
        out = 0;        // 0 < result <= sigma
}
else
{
```

```
    // negative outputs
    if( result < sigma )
        out = -2;      // smallest output
    else
        out = -1;      // sigma <= result <= 0
}
```

Note that the inner **if-else** statements are compound statements (each consisting of two statements), which make the braces necessary in the outer **if-else** control structure (without the braces there would be too many **else** statements resulting in a compilation error).

2.4.2 The switch Statement

When a program must choose between several alternatives the **if-else** statement becomes inconvenient and sometimes inefficient. When more than four alternatives from a single expression are chosen, the **switch** statement is very useful. The basic form of the **switch** statement is as follows:

```
switch( integer expression )
{
    case constant1:
        statements;     (optional)
        break;          (optional)
    case constant2:
        statements;     (optional)
        break;          (optional)
        . . . . .       (more optional statements)
    default:            (optional)
        statements;     (optional)
}
```

Program control jumps to the statement after the case label that has the constant (an integer or single character in quotes) matches the result of the integer expression in the **switch** statement. If no constant matches the expression value, control goes to the statement following the default label. If the default label is not present and no matching case labels are found, then control proceeds with the next statement following the **switch** statement. When a matching constant is found, the remaining statements after the corresponding case label are executed until the end of the switch statement is reached or a **break** statement is reached that redirects control to the next statement after the **switch** statement. A simple example is as follows:

```
switch( i )
{
    case 0:
        cout << "Error: I is zero\n";
```

```
            break;
        case 1:
            j = k * k;
            break;
        default:
            j = k * k / i;
    }
```

The use of the **break** statement after the first two case statements is required in order to prevent the next statements from being executed (a **break** is not required after the last **case** or **default** statement). Thus, the above code segment sets **j** equal to **k * k / i** unless **i** is zero, in which case it will indicate an error and leave **j** unchanged. Note that since the divide operation usually takes more time than the case statement branch, some execution time will be saved whenever **i** equals 1.

2.4.3 Single-Line Conditional Expressions

One **if-else** control structure can be expressed in a single line. It is called a *conditional expression* because it uses the conditional operator, **?:,** which is the only trinary operator. The general form of the conditional expression is

```
expression1 ? expression2 : expression3
```

If **expression1** is true (nonzero), then the whole conditional expression has the value of **expression2**. If **expression1** is false (0), the whole expression has the value of **expression3**. One simple example is finding the maximum of two expressions as follows:

```
maxdif = ( a0 > a2 ) ? a0 - a1 : a2 - a1;
```

Conditional expressions are not necessary, since **if-else** statements can provide the same function. Conditional expressions are more compact and sometimes lead to more efficient machine code. On the other hand, they are often more confusing than the familiar **if-else** control structure.

2.4.4 Loops: `while`, `do-while`, and `for`

There are three control structures that allow a statement or group of statements to be repeated a fixed or variable number of times. The **while** loop repeats the statements until a test expression becomes false, or zero. The decision to go through the loop is made before the loop is ever started. Thus, it is possible that the loop is never traversed. The general form is

```
while( expression )
    statement
```

where statement can be a single statement or a compound statement enclosed in braces. An example of the latter that counts the number of spaces in a null-terminated string (an array of characters) is as follows:

```
int space_count = 0;     // space_count is an int
int i = 0;               // array index, i = 0
while( string[i] )
{
    if( string[i] == ' ' )
        space_count++;
    i++;                 // next char
}
```

Note that if the string is zero length, then the value of **string[i]** will initially point to the null terminator (which has a zero or false value) and the **while** loop will not be executed. Normally, the **while** loop will continue counting the spaces in the string until the null terminator is reached.

The **do-while** loop is used when a group of statements needs to be repeated and the exit condition should be tested at the end of the loop. The decision to go through the loop one more time is made after the loop is traversed so that the loop is always executed at least once. The format of **do-while** is similar to the **while** loop except that the **do** keyword starts the statement and **while(expression)** ends the statement. A single or compound statement may appear between the do and the while keywords. A common use for this loop is in testing the bounds on an input variables as the following example illustrates:

```
do
{
    cout << "Enter FFT length (less than 1025) : ";
    cin >> fft_length;
}
while( fft_length > 1024 );
```

In this code segment, if the integer **fft_length** entered by the user is larger than 1024, the user is prompted again until the **fft_length** entered is 1024 or less.

The **for** loop combines an initialization statement, an end condition statement, and an action statement (executed at the end of the loop) into one very powerful control structure. The standard form is

```
for( initialize ; test condition ; end update )
    statement;
```

The three expressions are all optional [**for(;;);** is an infinite loop] and the statement may be a single statement, a compound statement or just a semicolon (a null statement). The most frequent use of the for loop is indexing an array through its elements. For example,

```
for( int i = 0; i < length; i++ )
    a[i] = 0;
```

sets the elements of the array **a** to zero from **a[0]** up to and including **a[length-1]**. This **for** statement sets **i** to zero and checks to see if **i** is less than **length**; if so it executes the statement **a[i] = 0;**, increments **i**, and then repeats the loop until **i** is equal to **length**. The integer **i** is incremented or updated at the end of the loop, and then the test condition statement is executed. Thus, the statement after a **for** loop is only executed if the test condition in the **for** loop is true. **For** loops can be much more complicated because each statement can be multiple expressions as the following example illustrates:

```
for( i = 0, i3 = 1; i < 25; i++, i3 = 3 * i3)
    cout << i << " " << i3 << endl;
```

This statement uses two **int**s in the **for** loop **(i, i3)** to print the first 25 powers of 3. Note that the end condition is still a single expression **(i < 25)** but that the initialization and end expressions are two assignments for the two integers separated by a comma.

2.4.5 Program Jumps: `break`, `continue`, and `goto`

The loop control structures just discussed and the conditional statements (**if, if-else,** and **switch**) are the most important control structures. They should be used exclusively in the majority of programs. The last three control statements (**break**, **continue**, and **goto**) allow for conditional program jumps. If used excessively, they will make a program harder to follow, more difficult to debug, and harder to modify.

The **break** statement, which was already illustrated in conjunction with the switch statement, causes the program flow to break free of the **switch**, **for**, **while**, or **do-while** that encloses it and proceed to the next statement after the associated control structure. Sometimes **break** is used to leave a loop when there are two or more reasons to end the loop. Usually, however, it is much clearer to combine the end conditions in a single logical expression in the loop test condition. The exception to this is when a large number of executable statements are contained in the loop and the result of some statement should cause a premature end of the loop (e.g., an end of file or other error condition).

The **continue** statement is almost the opposite of **break;** the **continue** causes the rest of an iteration to be skipped and the next iteration to be started. The **continue** statement can be used with **for**, **while**, and **do-while** loops but cannot be used with **switch**. The flow of the loop in which the **continue** statement appears is interrupted, but the loop is not terminated. Although the **continue** statement can result in very hard-to-follow code, it can shorten programs with nested **if-else** statements inside one of three loop structures.

The **goto** statement is available even though it is never required. Most programmers with a background in FORTRAN or BASIC computer languages (both of which require

the goto for program control) have developed bad programming habits that make them depend on the goto. The **goto** statement uses a label rather than a number making things a little better. For example, one possible legitimate use of goto is for consolidated error detection and cleanup as the following simple example illustrates:

```
.
.
program statements
.
.
status = function_one( alpha, beta, constant );
if( status != 0 ) goto error_exit;
.
.
more program statements
.
.
status = function_two( delta, time );
if( status != 0 ) goto error_exit;
.
.
.
error_exit:        // end up here from all errors
    switch( status )
    {
    case 1:
        cerr << "Divide by zero error\n";
        break;
    case 2:
        cerr << "Out of memory error\n";
        break;
    case 3:
        cerr << "Log overflow error\n";
        break;
    default:
        cerr << "Unknown error\n";
        break;
    }
    exit();
```

In the above example, both of the fictitious functions, **function_one** and **function_two** (see the next section concerning the definition and use of functions), perform some set of operations that can result in one of several errors. If no errors are detected, the function returns zero and the program proceeds normally. If an error is detected, the integer **status** is set to an error code and the program jumps to the label **error_exit** where a message in-

dicating the type of error is printed before the program is terminated. Instead of using a **goto** statement that tests the error condition, the preferred method in C++ is to handle error conditions using exception handling as described next.

2.4.6 Exception Handling

Robust programs must handle innumerable errors at any point in their execution. Exception handling is a mechanism that allows programmers to specify the type of error handled and code to handle that error. When an exception is raised by the **throw** statement, the flow of control is transferred from the point where the exception occurred to a nonlocal point specified by the programmer as the exception handler. Exception handling provides an alternative to returning function error codes that are often ignored or unchecked leading to programs that are not robust or display undesired behavior. If a portion of code failed several function calls deep, each function on the call stack would have to explicitly check for any errors, then either return the error code or call the appropriate code to handle those errors.

The method of specifying the flow of control after an error occurs is to enclose the code in a *try* block and handle the exception in the *catch* clause. The following simple example illustrates the use of an exception handler with the **throw** statement:

```
#include <iostream.h>
#include <string.h>

void printLine( const char *line )
{
    // Validate parameters
    if( line == NULL )
        throw "NULL name passed into printLine()";
    cout << "The rest of the line is: " << line << endl;
}

void main()
{
    try
    {
        const char *str = "First Second Third";
        // Okay
        printLine( strstr( str, "First" ) );
        printLine( strstr( str, "Second" ) );
        printLine( strstr( str, "Third" ) );
        // NULL - error
        printLine( strstr( str, "Fourth" ) );
    }
    catch( const char *e )
    {
        cerr << "An error occurred in the program:\n";
```

```
            cerr << e << endl;
        }
    }
```

When running the program, the first three **printLine** statements will execute properly, but in the fourth one, the function **strstr** (from the standard string library discussed in Appendix A) will not find the string "Fouth" and will return **NULL** for the pointer to the string. Since the program doesn't explicitly check each return parameter, this **NULL** pointer gets passed along to the **printLine** function, which will dereference a NULL pointer when trying to print it out. If the **printLine** function returned an error code instead of raising an exception, the main routine would then also have to check for this rare case and jump to the appropriate section of code to handle the error.

In this simple example, it is easy to see many ways to avoid this error, but it becomes more difficult as the program becomes more complex and there are many more errors that can occur. If used correctly, exception handling provides a clean mechanism to ensure that errors are not ignored.

Since the flow of control is passed to a nonlocal point, exception handling should not be used for return conditions that happen under the normal execution of the program because performance and readability of the code would be reduced.

2.5 FUNCTIONS

All programs consist of one or more functions. Even the program executed first is a function called **main()** as illustrated in Listing 2.1. Thus, unlike other programming languages, there is no distinction between the "main" program and programs that are called by the main program (sometimes called *subroutines*). A function may or may not return a value, thereby removing another distinction between subroutines and functions in languages such as FORTRAN. Each function is a program equal to every other function. Any function can call any other function (a function can even call itself) or be called by any other function. This makes C and C++ functions somewhat different from Pascal procedures, where procedures nested inside one procedure are ignorant of procedures elsewhere in the program. It should also be pointed out that unlike FORTRAN and several other languages, C and C++ always pass functions arguments "by value" not "by reference." Because arguments are passed by value, when a function must modify a variable in the calling program, the programmer must specify the function argument as a pointer to the beginning of the variable in the calling program's memory. (See Section 2.7 for a discussion of pointers.)

2.5.1 Defining and Declaring Functions

A function is defined by the function type, a function name, a pair of parentheses containing an optional formal argument list, and a pair of braces containing the optional executable statements. The general format is as follows:

```
type name( formal argument list )
{
    function body
}
```

The **type** determines the type of value the function returns, not the type of arguments. If no **type** is given, the function is assumed to return an **int** (actually, a variable is also assumed to be of type **int** if no type specifier is provided). If a function does not return a value, it should be declared with the type **void**. For example, Listing 2.1 contains the function average as follows:

```
float average( const float *array, int size )
{
    if( array == NULL || size == 0 )      // validate parameters
        throw "Invalid parameters";
    float sum = array[0];                 // initialize sum
    for( int i = 1; i < size; i++ )
        sum = sum + array[i];             // calculate sum
    return( sum / size );                 // return average
}
```

The first line in the above code segment, **float average(const float *array, int size),** declares a function called *average* will return a single-precision floating-point value and will except two arguments. The two *argument names* (**array** and **size**) are defined in the formal *argument list* (also called the formal parameter list). The types of the two arguments specify that **array** is a pointer to an array of **floats** that cannot be modified and **size** is an **int**. The body of the function that defines the executable statements and local variables to be used by the function are contained between the two braces. Before the ending brace (}), a return statement is used to return the **float** result back to the calling program. If the function did not return a value (in which case it should be declared **void**), simply omitting the return statement would return control to the calling program after the last statement before the ending brace. When a function with no return value must be terminated before the ending brace (if an error is detected, for example), a **return;** statement without a value should be used. The parentheses following the return statement are only required when the result of an expression is returned. Otherwise, a constant or variable may be returned without enclosing it in parentheses (e.g., **return 0;** or **return n;**).

If either of the input parameters are invalid (there is no data or the size of the data is zero), the throw command would return control to the exception handler for the **const char *** type at the end of the **try** block.

Note that the variable **array** is actually just a pointer to the beginning of the **float** array that was allocated by the calling program. By passing the pointer, only *one value* is passed to the function and not the large floating-point array. In fact, the function could also be declared to pass an array of unknown size as follows:

```
float average( const float array[], int size )
```

Arguments are used to convey values from the calling program to the function. Because the arguments are passed by value, a local copy of each argument is made for the function to use (usually the variables are stored on the stack by the calling program). The local copy of the arguments may be freely modified by the function body but will not change the values in the calling program, since only the copy is changed. The return statement can communicate one value from the function to the calling program. Other than this returned value, the function may not directly communicate back to the calling program. This method of passing arguments "by value" such that the calling program's variables are isolated from the function avoids the common problem in FORTRAN, where modifications of arguments by a function get passed back to the calling program, resulting in the occasional modification of constants within the calling program.

When a function must return more than one value, one or more pointer arguments must be used. The calling program must allocate the storage for the result and pass the function a pointer to the memory area to be modified. The function then gets a copy of the pointer that it uses (with the indirection operator, ***,** discussed in more detail in Section 2.7.1) to modify the variable allocated by the calling program. For example, the functions **average** and **variance** in Listing 2.1 can be combined into one function that passes the arguments back to the calling program in two **float** pointers called **ave** and **var,** as follows:

```
void stats( const float *array, int size, float *ave, float *var )
{
    if( array == NULL || ave == NULL || var == NULL || size < 2 )
        throw "Invalid parameters";
    float sum = 0.0f;                    // initialize sum
    float sum2 = 0.0f;                   // signal squared sum
    for( int i = 0; i < size; i++ )
    {
        sum = sum + array[i];            // calculate sums
        sum2 = sum2 + array[i] * array[i];
    }
    *ave = sum / size;                   // average and variance
    *var = ( sum2 - sum * (*ave) ) / ( size - 1 );
}
```

In this function, no value is returned, so it is declared type **void** and no return statement is used. Since the **ave** and **var** are now pointers to memory and will be dereferenced, they are validated at the beginning of the function and an exception will be raised if they are invalid. The **size** parameter must be greater than one to avoid a divide-by-zero error. This **stats** function is more efficient than the functions **average** and **variance** together because the sum of the array elements was calculated by both the average function and the variance function. If the variance is not required by the calling program, then the average function alone is much more efficient because the sum of the squares of the array elements is not required to determine the average alone.

When calling the **stats** function, the **main** routine would specify the address for the **ave** and **var** variables with the **&** operator as

```
float ave, var;
stats( signal, count, &ave, &var );
```

The **&** operator is discussed later in Section 2.7.1.

2.5.2 Storage Class, Privacy, and Scope

In addition to type, variables and functions have a property called *storage class.* There are four storage classes with four storage class designators: **auto**, for *automatic variables* stored on the stack; **extern**, for *external variables* stored outside the current module; **static**, for variables known *only in the current module;* and **register**, for *temporary variables* to be stored in one of the registers of the target computer. Each of these four storage classes defines the scope or degree of the privacy a particular variable or function holds. The storage class designator keyword (**auto**, **extern**, **static**, or **register**) must appear first in the variable declaration before any type specification. The privacy of a variable or function is the degree to which other modules or functions cannot access a variable or call a function. Scope is, in some ways, the complement of privacy because the scope of a variable describes how many modules or functions have access to the variable.

Auto variables can be declared only within a function, are created when the function is invoked, and are lost when the function is exited. **Auto** variables are known only to the function in which they are declared and do not retain their value from one invocation of a function to another. Because **auto** variables are stored on a stack, a function that uses only **auto** variables can call itself recursively. The **auto** keyword is rarely used, since variables declared within functions default to the **auto** storage class.

Another important distinction of the **auto** storage class is that an **auto** variable is defined only within the control structure that surrounds it. That is, the scope of an **auto** variable is limited to the expressions between the braces (**{** and **}**) containing the variable declaration. For example, the following simple program would generate a compiler error, since **j** is unknown outside of the **for** loop:

```
#include <iostream.h>

void main()
{
    for( int i = 0; i < 10; i++ )
    {
        int j;                              // declare j here
        j = i * i;
        cout << j << endl;                  // j known here
    }
    cout << j << endl;                      // Error: j unknown
}
```

Register variables have the same scope as **auto** variables but are stored in some type of register in the target computer. If the target computer does not have registers or if no more registers are available in the target computer, a variable declared as **register** will revert to **auto**. Because almost all microprocessors have a large number of registers that can be accessed much faster than outside memory, **register** variables can be used to speed up program execution significantly. Most compilers limit the use of **register** variables to pointers, integers, and characters because the target machines rarely have the ability to use registers for floating-point or double-precision operations.

Extern variables have the broadest scope. They are known to all functions in a module and are even known outside of the module in which they are declared. **Extern** variables are stored in their own separate data area and must be declared outside of any functions. Functions that access **extern** variables must be careful not to call themselves or call other functions that access the same **extern** variables, since **extern** variables retain their value as functions are entered and exited. **Extern** is the default storage class for variables declared outside of functions and for the functions themselves. Thus, functions not declared otherwise may be invoked by any function in a module as well as by functions in other modules.

Static variables differ from **extern** variables only in scope. A **static** variable declared outside of a function in one module is known only to the functions in that module. A **static** variable declared inside a function is known only to the function in which it is declared. Unlike an **auto** variable, a **static** variable retains its value from one invocation of a function to the next. Thus, **static** refers to the memory area assigned to the variable and does not indicate that the value of the variable cannot be changed. Functions may also be declared **static,** in which case the function is known only to other functions in the same module. In this way, the programmer can prevent other modules (and, thereby, other users of the object module) from invoking a particular function.

2.5.3 Function Prototypes

A function prototype is a statement (which must end with a semicolon) describing a particular function. It tells the compiler the type of the function (i.e., the type of the variable it will return) and the type of each argument in the formal argument list. The function named in the function prototype may or may not be contained in the module where it is used. All compilers provide a series of header files that contain the function prototypes for all of the standard functions (see Appendix A for a detailed description of the standard header files). For example, the prototype for the stats function defined in Section 2.5.1 is as follows:

```
void stats( const float *array, int size, float *ave, float *var );
```

This prototype indicates that **stats** (which can be in another module) returns no value and takes four arguments. The first argument is a pointer to the array of **floats** (in this case, the array to do statistics on). The second argument is an integer (in this case, giving the size of the array), and the last two arguments are pointers to **float**s where the **stats** function will return the average and variance results.

The result of using function prototypes for all functions used by a program is that the compiler now knows what type of arguments are expected by each function. This information can be used in different ways. Some compilers convert whatever type of actual argument is used by the calling program to the type specified in the function prototype and issue a warning that a data conversion has taken place. Other compilers simply issue a warning indicating that the argument types do not agree and assume that the programmer will fix it if such a mismatch is a problem.

2.5.4 Templates

Templates offer a mechanism to create functions for many types without explicitly naming the types and without giving up type checking. They are used in many places throughout the C++ code to allow the **Vector** and **Matrix** classes to work on any system type or user-defined class. Since the **Vector** and **Matrix** classes operate on an array of data and do not need to know the type of data they are managing, they are good candidates for templates. The syntax for templates is to use the template keyword followed by formal type parameters between **<** and **>**.

Templates also can be used on functions that are designed to operate on different types as illustrated by the following example:

```
template <class Type>
const Type& max( const Type& t1, const Type& t2 )
{
    return( ( t1 > t2 ) ? t1 : t2 );
}
```

This templated function takes two *references* (see Section 2.7) of type **Type** and returns the reference that is the greater of the two. This will work for all standard types and for classes that, in this case, overload the > operator as described in Section 2.3.5.

2.6 MACROS AND THE C PREPROCESSOR

The *preprocessor* is one of the most useful features of the C and C++ programming languages. Although most languages allow compiler constants to be defined and used for conditional compilation, few languages (except for assembly language) allow the user to define macros. The large set of preprocessor directives can be used to completely change the look of a program such that it is very difficult for anyone to decipher. On the other hand, the preprocessor can be used to make complicated programs easy to follow, very efficient, and easy to code. The remainder of this chapter and the programs discussed in this book hopefully will serve to illustrate the latter advantages of the preprocessor.

The preprocessor allows *conditional compilation* of program segments, *user-defined symbolic replacement* of any text in the program, and *user-defined multiple parameter macros.* All of the preprocessor directives are evaluated before any code is compiled, and the directives, themselves, are removed from the program before compilation

begins. Each preprocessor directive begins with a pound sign (#) followed by the preprocessor keyword. The following list indicates the basic use of each of the most commonly used preprocessor directives:

#define NAME macro	associate symbol **NAME** with **macro** definition (optional parameters)
#include "file"	copy named **file** (with directory specified) into current compilation
#include <file>	include **file** from standard library
#if expression	conditionally compile the following code if result of **expression** is true
#ifdef symbol	conditionally compile the following code if the **symbol** is defined
#ifndef symbol	conditionally compile the following code if the **symbol** is not defined
#else	conditionally compile the following code if the associated **#if** is not true
#endif	indicates the end of previous **#else, #if, #ifdef** or **#ifndef**
#undef macro	undefine previously defined **macro**

2.6.1 Conditional Preprocessor Directives

Most of the above preprocessor directives are used for conditional compilation of portions of a program. For example, in the following version of the **stats** function (described previously in Section 2.5.1) the definition of **DEBUG** is used to indicate that the print statements should be compiled:

```
void stats( const float *array, int size, float *ave, float *var )
{
    if( array == NULL || ave == NULL || var == NULL || size < 2 )
         throw "Invalid parameters";
    float sum = 0.0f;                    // initialize sum
    float sum2 = 0.0f;                   // signal squared sum
    for( int i = 0; i < size; i++ )
    {
        sum = sum + array[i];            // calculate sums
        sum2 = sum2 + array[i] * array[i];
    }
#ifdef DEBUG
    cerr << "In stats sum = " << sum << " sum2 = " << sum2 << endl;
    cerr << "Number of array elements = " << size << endl;
     #endif // DEBUG
         *ave = sum / size;                    // average and variance
         *var = ( sum2 - sum * (*ave) ) / ( size - 1 );
     }
```

If the preprocessor parameter **DEBUG** is defined anywhere before the **#ifdef DEBUG** statement, then the output statements to **cerr** will be compiled as part of the program to aid in debugging **stats** (or perhaps even the calling program). Many compilers allow the definition of preprocessor directives when the compiler is invoked by either the command line or inside of the visual programming environment. This allows the **DEBUG** option to be used with no changes to the program text.

2.6.2 Macros

Of all the preprocessor directives, the **#define** directive is the most powerful because it allows multiple parameter macros to be defined in a relatively simple way. The most common use of **#define** is a macro with no arguments that replaces one string (the macro name) with another string (the macro definition). In this way, a macro can be given to any string including all of the language keywords. For example:

```
#define DO for(
```

replaces every occurrence of the string **DO** (all capital letters so that it is not confused with the language keyword **do**) with the four-character string **for**(. Similarly, new macros of all the language keywords could be created with several **#define** statements (although this seems silly, since the normal keywords seem good enough). Even single characters can be **#defined**. For example, **BEGIN** could be **{** and **END** could be **}** which makes a C or C++ program look more like Pascal.

The **#define** directive is much more powerful when parameters are used to create a true macro. The above **DO** macro can be expanded to define a simple FORTRAN style DO loop as follows:

```
#define DO( var, beg, end ) for( var = beg; var <= end; var++ )
```

The three macro parameters **var**, **beg**, and **end** are the variable, the beginning value, and the ending value of the DO loop. In each case, the macro is invoked and the string placed in each argument is used to expand the macro. For example:

```
int i;
DO( i,1,10 )

expands to

for( i = 1; i <= 10; i++ )
```

which is the valid beginning of a **for** loop that will start the variable i at 1 and stop it at 10. Although this DO macro does shorten the amount of typing required to create such a simple **for** loop, it must be used with caution. When macros are used with other operators, other macros, or other functions, unexpected program bugs can occur. For example,

the above macro will not work at all with a pointer as the **var** argument, because **DO(*ptr,1,10)** would increment the pointer's value and not the value it points to (see Section 2.7.1). This would probably result in very strange number of cycles through the loop (if the loop ever terminated). As another example, consider the following **CUBE** macro, which will determine the cube of a variable:

```
#define CUBE(x) (x)*(x)*(x)
```

This macro will work fine (although inefficiently) with **CUBE(i+j)**, since it would expand to **(i+j)*(i+j)*(i+j)**. However, **CUBE(i++)** expands to **(i++)*(i++)*(i++),** resulting in **i** getting incremented three times instead of once, and the resulting value would be i*(i+1)*(i+2) not **i** cubed.

The ternary conditional operator (see Section 2.4.3) can be used with macro definitions to make fast implementations of the absolute value of a variable **(ABS)**, the minimum of two variables **(MIN),** the maximum of two variables **(MAX),** and the integer-rounded value of a floating-point variable **(ROUND)** as follows:

```
#define ABS(a)      (((a) < 0) ? (-a) : (a)
#define MAX(a,b)    (((a) > (b)) ? (a) : (b))
#define MIN(a,b)    (((a) < (b)) ? (a): (b))
#define ROUND(a)    (((a)<0)?(int)((a)-0.5):(int)((a)+0.5))
```

Note that each of the above macros is enclosed in parenthes so that it can be used freely in expressions without uncertainty about the order of operations. Parentheses are also required around each of the macro parameters, since these may contain operators as well as simple variables.

All of the macros defined so far have names that contain only capital letters. While this is not required, it does make it easy to separate macros from normal keywords in programs where macros may be defined in one module and included (using the **#include** directive) in another. This practice of capitalizing all macro names and using lower case for variable and function names will be used in all programs in this book and on the accompanying disk.

2.6.3 Inline Functions

The ability to create macros with the **#define** preprocessor directive is very powerful but has several drawbacks:

1. The macro simply replaces parameters without any type checking.
2. Hard-to-find errors can be introduced by errors in the macro or by evaluation side effects of parameters.

3. Ambiguity is formed when the macro replaces common programming constructs like the **DO(var,beg,end)** mentioned above.
4. Macros that compare two variables may compare two different types without the compiler warning the programming.

The output of the following program will demonstrate these problems:

```
#include <iostream.h>

#define MAX(a,b)    (((a) > (b)) ? (a) : (b))

void main()
{
    int a = 0;
    int b = 1;

    // Returns 2 instead of 1
    cout << MAX( a, b++ ) << endl;
}
```

The use of the **inline** keyword before the function return type solves these problems. Inlined functions may be inserted into the code by the compiler for an optimization but have formal parameters and type checking. The use of templates, described earlier in Section 2.5.4, allows the function to accept multiple types but validate the arguments, for example:

```
#include <iostream.h>

template <class Type>
inline const Type& max( const Type& t1, const Type& t2 )
{
    return( ( t1 > t2 ) ? t1 : t2 );
}
void main()
{
    int a = 0;
    int b = 1;

    // Returns 1 correctly
    cout << max( a, b++ ) << endl;
}
```

Several commonly used functions are provided as inline function in the **dsp.h** file. They are **min**(), **max**(), **round**(), **uniform**(), **gaussian**(), and **log2**().

2.6.4 Constant Variables

The are some advantages of defining constant variables by using the **const** keyword. The main benefits of this are giving the variable a type so that the compiler can perform better type checking, evaluating the constant once, and allowing symbolic debuggers to display the value of the variable while debugging. For example, the value for π could be defined with the **#define** preprocessor directive as in

```
#define PI ( 4.0 * atan( 1.0 ) )
```

The problem is that every time **PI** is used in a program, the expression **(4.0*atan(1.0))** is inserted into the expression and reevaluated. In most visual programming environments, unfortunately, the debugger does not know what value to display, making incorrect constants hard to find. The solution is to have the compiler explicitly specify the type and evaluate the expression with the statement

```
const double PI = 4.0 * atan( 1.0 );
```

which is evaluated by the compiler and stored in the constant variable **PI**.

2.7 POINTERS, ARRAYS, AND REFERENCES

A *pointer* is a variable that holds an address of some data rather than the data itself. The use of pointers is usually closely related to manipulating (assigning or changing) the elements of an array of data. Pointers are used primarily for three purposes:

1. To point to different data elements within an array
2. To allow a program to create new variables while a program is executing (dynamic memory allocation)
3. To access different locations in a data structure

The first two uses of pointers will be discussed in this section; pointers to data structures are considered in Section 2.8.4.

A *reference* serves as an alternate name of an object that has been instantiated. All operations on the reference act upon the object to which it refers. References are very similar to pointers in usage, except that they must alias an existing object (there are no NULL references) and that once they are initialized, they cannot be changed to alias a different object. References are commonly associated with input parameters to a function. Some examples are listed in Section 2.7.4.

2.7.1 Special Pointer Operators

Two *special pointer operators* are required to effectively manipulate pointers: the indirection operator (*) and the address of operator (**&**). The indirection operator (*****) is used whenever the data stored at the address pointed to by a pointer is required, that is, whenever indirect addressing is required. Consider the following simple program:

```
#include <iostream.h>

void main()
{
    int i = 7;                              // set the value of i
    int *ptr = &i;                          // point to address of i
    cout << i << endl;                      // print i two ways
    cout << *ptr << endl;
    *ptr = 11;                              // change i with pointer
    cout << *ptr << " " << i << endl;      // print change
}
```

This program declares that **i** is an integer variable set to 7 and that **ptr** is a pointer to an integer variable set to the address of **i** by the statement **int ptr = &i;**. The compiler assigns **i** and **ptr** storage locations somewhere in memory. At run time, **ptr** is set to the starting address of the integer variable **i**. The above program uses the **cout** class to display the integer value of **i** in two different ways—by printing the contents of the variable **i** in **cout << i << endl;** and by using the indirection operator **cout << *ptr << endl;**. The presence of the * operator in front of **ptr** directs the compiler to pass the value stored at the address **ptr** to the **cout** class (in this case, 7). If only **ptr** were used, then the address assigned to **ptr** would be displayed instead of the value 7. The last two lines of the example illustrate indirect storage; the data at the **ptr** address is changed to 11. This results in changing the value of **i** only because **ptr** is pointing to the address of **i**.

An *array* is essentially a section of memory that is allocated by the compiler and assigned the name given in the declaration statement. In fact, the name given is nothing more than a fixed pointer to the beginning of the array. The array name can be used as a pointer, or it can be used to reference an element of the array (i.e., **a[2]**). If **a** is declared as some type of array, then ***a** and **a[0]** are exactly equivalent. Furthermore, ***(a+i)** and **a[i]** are also the same (as long as **i** is declared as an integer), although the meaning of the second is often more clear. Arrays can be rapidly and sequentially accessed by using pointers and the increment operator (**++**). For example, the following three statements set the first 100 elements of the array **a** to 10:

```
int a[100];
int *pointer = a;
for( int i = 0; i < 100; i++ )
    *pointer++ = 10;
```

On many computers this code will execute faster than the single statement **for(int i = 0; i < 100; i++) a[i] = 10;** because the post increment of the pointer is faster than the array index calculation, which requires a multiplication of the type size and the index and the addition of the base of the array.

2.7.2 Pointers and Dynamic Memory Allocation

There are two operators in C++ that allow the programmer to dynamically change the type and size of variables and arrays of variables stored in the computer's memory. Programs can use the same memory for different purposes and not waste large sections of memory on arrays that are used only in one small section of a program. In addition, **auto** variables are automatically allocated on the stack at the beginning of a function (or any section of code where the variable is declared within a pair of braces) and removed from the stack when a function is exited (or at the right brace, }). By proper use of **auto** variables (see Section 2.5.2) and the dynamic memory allocation functions, the memory used by a particular program can be very little more than the memory required by the program at every step of execution. This feature is especially attractive in multi-user environments where the product of the memory size required by a user and the time that memory is used ultimately determines the overall system performance. In many DSP applications, the proper use of dynamic memory allocation can enable a complicated DSP function to be performed with an inexpensive single-chip signal processor with a small limited internal memory size instead of a more costly processor with a larger external memory.

Two standard operators are used to manipulate the memory available to a particular program. **New** allocates storage and **delete** removes a previously allocated item from the memory pool associated with a program.

When using the **new** operator, the type and size of the item to be allocated must be passed to the operator. The **new** operator then initializes an array of elements of the type specified and returns a pointer of that type to a block of memory. The type allocated can be any of the standard types provided by the language as well as user-defined types as shown below:

```
// Define number of students
const int STUDENTS = 64;

// Allocate an array of 64 ints
int *grades = new int[STUDENTS];

// Allocate an array of objects of type String
String *names = new String[STUDENTS];

// Create a single object
String *teacher = new String;
```

The first **new** operator allocates storage for an array of integers and points the integer pointer, **grades,** to the beginning of the memory block. On 32-bit machines, the bytes allocated by this statement will be 4 bytes (or one word) multiplied by the number inside the brackets (64 in this example). The next allocation with the **new** operator allocates and initializes an array of 64 String objects and assigns them to the **names** pointer. The last example allocates and initializes a single String object and points the pointer **teacher** to it. Memory allocation for classes and their initialization using constructors will be discussed in Section 2.8.3.

The arrays can then be referenced by using another pointer (changing the pointer array is unwise, since it holds the position of the beginning of the allocated memory) or by an array reference such as **grades[i]** (where **i** may be from 0 to 63). The memory block allocated by **new** is not initialized and may contain random values.

New and **delete** provide a simple general-purpose memory allocation package. The argument to **delete** is a pointer to an object or block of memory previously allocated by **new**; this space is made available for further allocation, but its contents are left undisturbed. To release the memory allocated above, each pointer must be **deleted** as follows:

```
// Release the grades array
delete [] grades;

// Release the names array;
delete [] names;
// Release a single object
delete teacher;

// NULL pointers to make sure nobody uses them
grades = NULL;
names = NULL;
teacher = NULL;
```

Since **grades** and **names** are arrays, allocated by a size between brackets, they are released with empty brackets, telling the compiler to perform any destruction on the array elements. It is good practice to set released pointers equal to **NULL** to ensure that they are not used in the rest of the program until reallocated. Deleting a **NULL** pointer as in

```
// Initialize names and number of students
String *names = NULL;

// Release NULL memory
delete [] names;
```

is not an error and will continue executing the next statement.

2.7.3 Arrays of Pointers

Any of the data types or pointers to each of the data types can be declared as an array. Arrays of pointers are especially useful in accessing large matrices. An array of pointers to 10 rows each of 20 integer elements can be dynamically allocated as follows:

```
void main()
{
    // Declare a pointer to an array of int pointers
    int **mat = NULL;

    // Allocate an array of 10 int pointers
    mat = new int*[10];
    if( mat == NULL )
        throw "Error in matrix allocation";

    // Set each element in mat to an allocated array
    for( int i = 0; i < 10; i++ )
    {
        // Allocate an array of 20 ints
        mat[i] = new int[20];
        if( mat[i] == NULL )
            throw "Error in data allocation";
    }

    // Use matrix
    mat[5][6] = 10;
    cout << mat[5][6] << endl;
    mat[6][7] = mat[5][6] + 1;
    cout << mat[6][7] << endl;

    // Delete 10 arrays of 20 ints
    for( i = 0; i < 10; i++ )
        delete [] mat[i];

    // Delete pointer to array of 10 int pointers
    delete [] mat;
    mat = NULL;
}
```

In this code segment, the pointer to an array of integer pointers is declared and allocated, then each pointer is set to 10 different memory blocks allocated by 10 successive calls to **new**. After each call to **new,** the pointer must be checked to ensure that the memory was available **(mat[i]** will be allocated if **mat[i]** is non-null). Each element in the matrix **mat** can now be accessed by using pointers and the indirection operator. For example, ***(mat[i] + j)** gives the value of the matrix element at the **i**th row (0–9) and the **j**th column

(0–19) and is exactly equivalent to **mat[i][j]**. In fact, the above code segment is equivalent (in the way **mat** may be referenced at least) to the array declaration **int mat[10][20];** except that **mat[10][20]** is allocated as an **auto** variable on the stack and the above calls to **new** allocates the space for **mat** on the heap. Note, however, that when **mat** is allocated on the stack as an **auto** variable it cannot be used with **delete.** The calculations required by the compiler to access a particular element in a two-dimensional matrix (declared with **mat[10][20],** for example) usually take more instructions and more execution time than accessing the same matrix using pointers. This is especially true if many references to the same matrix row or column are required. However, depending on the compiler and the speed of pointer operations on the target machine, access to a two-dimensional array with pointers and simple pointers operands (even increment and decrement) may take almost the same time as a reference to a matrix such as **a[i][j]**. For example, the product of two 100 x 100 matrices could be coded using two-dimensional array references as follows:

```
// Declare three 100x100 matrices
int a[100][100], b[100][100], c[100][100];

// Code to set up mat goes here
// ...
// Do matrix multiply c = a * b
for( int i = 0; i < 100; i++ )
{
    for( int j = 0; j < 100; j++ )
    {
        c[i][j] = 0;
        for( int k = 0; k < 100; k++ )
            c[i][j] += a[i][k] * b[k][j];
    }
}
```

The same matrix product could also be performed using arrays of pointers as follows:

```
// Do matrix multiply c = a * b
for( int i = 0; i < 100; i++ )
{
    int *cptr = c[i];
    int *bptr = b[0];
    for( int j = 0; j < 100; j++ )
    {
        int *aptr = a[i];
        *cptr = (*aptr++) * (*bptr++);
        for( int k = 1; k < 100; k++ )
            *cptr += (*aptr++) * b[k][j];
        cptr++;
    }
}
```

The latter form of the matrix multiply code using arrays of pointers runs 10% to 20% faster depending on the degree of optimization done by the compiler and the capabilities of the target machine. Note that **c[i]** and **a[i]** are references to arrays of pointers each pointing to 100 integer values. Three factors help make the program with pointers faster:

1. Pointer increments (such as ***aptr++**) are usually faster than pointer adds.
2. No multiplies or shifts are required to access a particular element of each matrix.
3. The first add in the inner most loop (the one involving **k**) was taken outside the loop (using pointers **aptr** and **bptr**), and the initialization of **c[i][j]** to zero was removed.

2.7.4 References

References create an alias to an instantiated object and are most often used as parameters of functions. The **stats** function used in Section 2.5.1 declared the **ave** and **var** input parameters as pointers in order to modify them:

```
void stats( const float *array, int size, float *ave, float *var )
{
    if( array == NULL || ave == NULL || var == NULL || size < 2 )
        throw "Invalid parameters";

    float sum = 0.0f;                       // initialize sum
    float sum2 = 0.0f;                      // signal squared sum
    for( int i = 0; i < size; i++ )
    {
        sum = sum + array[i];               // calculate sums
        sum2 = sum2 + array[i] * array[i];
    }
    *ave = sum / size;                      // average and variance
    *var = ( sum2 - sum * (*ave) ) / ( size - 1 );
}
```

The calling function would specify the address of the variable as

```
stats( signal, count, &ave, &var );
```

The address of operator (**&**), discussed in Section 2.7.1, takes the address of **ave** and **var** and passes them to the **stats** function. However, since pointers can point to any memory location (even **NULL**), nothing prevents the calling routine from passing in a null memory address, as in

```
float *ptrave = NULL;
stats( signal, count, ptrave, &var );
```

The pointer **ptrave** doesn't have any memory assigned to it, but the code above is legal. Because objects of references must exist, the programmer can ensure that parameters exist before the function is used. The **stats** routine can be rewritten with references instead of pointers as follows:

```
void stats( const float *array, int size, float& ave, float& var )
{
    if( array == NULL || size < 2 )
        throw "Invalid parameters";

    float sum = 0.0f;                       // initialize sum
    float sum2 = 0.0f;                       // signal squared sum
    for( int i = 0; i < size; i++ )
    {
        sum = sum + array[i];               // calculate sums
        sum2 = sum2 + array[i] * array[i];
    }
    ave = sum / size;                       // average and variance
    var = ( sum2 - sum * ave ) / ( size - 1 );
}
```

The calling function would call this function as follows:

```
stats( signal, count, ave, var );
```

No **&** is used in the function call statement and no ***** is needed within the function when references are used. Using references for parameters still passes the memory location the same as pointers do, but now the parameters for **ave** and **var** must be **float** variables that exist. There is no need to make sure that they are non-null, because there can never be a reference to **NULL**. The **array** parameter is still a pointer because it addresses an array of **float**s. Since **ave** and **var** are changed at the bottom of this function, they are not declared **const** as **array** is. Passing a reference combines the efficiency of passing a pointer and the ease of use provided by passing a value. Although this simple example shows **float** references, aliases of C++ class objects are very common because the address of the class is passed to the function instead of a copy of the whole class.

2.8 STRUCTURES

Pointers and arrays allow the same type of data to be arranged in a list and easily accessed by a program. Pointers also allow arrays to be passed to functions efficiently and dynamically created in memory. When unlike data types that are logically related must be

manipulated, the use of several arrays becomes cumbersome. While it is always necessary to process the individual data types separately, it is often desirable to move all of the related data types as a single unit. The powerful data construct called a *structure* allows new data types to be defined as a combination of any number of the standard data types. Once the size and data types contained in a structure are defined (as described in the next section), the named structure may be used as any of the other data types. Arrays of structures, pointers to structures, and structures containing other structures may all be defined.

2.8.1 Declaring and Referencing Structures

A structure is defined by a structure template that indicates the type and name to be used to reference each element listed between a pair of braces. The general form of an N-element structure is as follows:

```
struct tag_name
{
    type1 element_name1;
    type2 element_name2;
            .
            .
            .
    typeN element_nameN;
} variable_name;
```

In each case, **type1**, **type 2**, ..., **typeN** refers to a valid data type (**char**, **int**, **float**, or **double** without any storage class descriptor) and **element_name1**, **element_name2**, ..., **element_nameN** refers to the name of one of the elements of the data structure. The **tag_name** is an optional name used for referencing the structure later. The optional **variable_name** or list of variable names define the names of the structures to be defined. The following structure template with a name of **RECORD** defines a structure containing an integer called **m_length**, a character array called **m_name**, and a pointer to an integer array called **m_data:**

```
struct RECORD
{
    int m_length;
    char m_name[80];
    int *m_data;
};
```

This structure template can be used to declare a structure called **voice** as follows:

```
RECORD voice;
```

The structure called **voice** of type **RECORD** can then be initialized as follows:

```
voice.m_length = 1000;
strcpy( voice.m_name, "voice signal" );
```

The **strcpy** function is needed because the string "voice signal" needs to be copied into the memory associated with **voice.m_name**. The standard library function **strcpy** copies all bytes from the array of bytes specified by the second parameter into the array of bytes specified by the first parameter until the end of the string is found. The **strcpy** function is described in Appendix A.

The last element of the structure is a pointer to the data and must be set to the beginning of a 1000-element integer array (because length is 1000 in the above initialization). Each element of the structure is referenced with the form **struct_name.element.** Thus, the 1000-element array associated with the voice structure can be allocated as follows:

```
voice.m_data = new int[1000];
```

Similarly, the elements of the structure can be displayed with the following code segment:

```
cout << "Length = " << voice.m_length << endl;
cout << "Record name = " << voice.m_name << endl;
```

2.8.2 Member Functions

One of the strongest features of C++ is the ability to declare functions that work on the data inside of a structure. This ability promotes the idea of *object-oriented* programming and *encapsulation,* where functions are combined with the data on which they operate. Functions inside of a structure are called member functions and are declared in the structure declaration as follows:

```
struct RECORD
{
    // Member variables
    int m_length;
    char m_name[80];
    int *m_data;

    // Member functions
    void setName( const char *name );
};
```

The function **setName** is a member function of the structure **RECORD**. It can access all variables inside of the structure without the member access operator (**.**), and the function can be defined elsewhere (most likely in the **.cpp** file) as

```
void RECORD::setName( const char *name )
{
    if( name == NULL )
        m_name[0] = 0;
    else
        strcpy( m_name, name );
}
```

The *scope operator* (**::**) between the structure name and function name tells the compiler that **setName** is a member function of the RECORD structure. If a program wanted to attach a name to the data, it could use the **setName** member function like this:

```
voice.setName( "Filtered voice signal" );
```

The advantage to having member functions work on the data is that routines outside of the structure no longer need to be concerned about the internal state of the structure. The ability to hide data inside a structure is a powerful method to improve the robustness of applications.

The body of the member function also can be defined within the structure itself without the use of the scope operator as follows:

```
struct RECORD
{
    // Member variables
    int m_length;
    char m_name[80];
    int *m_data;

  // Member functions
    void setName( const char *name )
    {
        if( name == NULL )
            m_name[0] = 0;
        else
            strcpy( m_name, name );
    }
};
```

2.8.3 Constructors and Destructors

C++ added the ability to initialize and clear data members inside of a structure on creation and deletion, respectively. The initialization routine is called the *constructor*. It is commonly used to initialize the structure to a known state before any operations are done

on it. The constructor is a member function that has the same name as the structure and does not return anything. An example constructor for the **RECORD** structure is

```
// Default Constructor
RECORD()
{
    setName( "Unknown data" );
    m_length = 0;
    m_data = NULL;
}
```

The constructor shown above initialized the member variables to a default state when the structure is created. It is good practice to initialize all member variables that are pointers to **NULL** on construction. Constructors do not return any values but can take parameters to initialize the structure to a state different from the default.

An example constructor that takes the name and length of the data, as input parameters, allocates memory for the data, and sets the member variables to the input parameters is as follows:

```
// Constructor
RECORD( const char *name, int length )
{
    m_data = new int[length];
    if( m_data == NULL )
        throw "Cannot allocate memory";

    m_length = length;
    setName( name );
}
```

The constructor attempts to allocate the memory first, and if successful, sets the member variables to the input parameters. There is a special form of a constructor called a *copy constructor*. The copy constructor takes an input parameter that is a reference to the same type as itself and initializes its member data from the reference. In the case of our **RECORD** structure, there is a problem if each element was bitwise copied directly from the reference because the pointer m_data would be copied instead of the data itself. When the original object was destroyed, the pointer would be deleted and any copies of the pointer become invalid. The solution to this is to explicitly implement the rules for creating objects from like objects. The copy constructor added to **RECORD** would allocate memory for the data, copy the data, then initialize the rest of the members. For this example, it will insert the words "copy of" in front of the name of the object being copied by copying the string using the standard library function **strcpy** followed by concatenating the name with the function **strcat:**

```
// Copy constructor
RECORD( const RECORD& rec )
{
    if( rec.m_data == NULL )
    {
        // No data
        m_data = NULL;
        m_length = 0;
    }
    else
    {
        // Data exists -- Create copy of rec
        m_data = new int[rec.m_length];
        if( m_data == NULL )
            throw "Cannot allocate memory";

        m_length = rec.m_length;
        for( int i = 0; i < m_length; i++ )
            m_data[i] = rec.m_data[i];
    }
    // Prepend "copy of " to beginning of name
    strcpy( m_name, "copy of " );
    strcat( m_name, rec.m_name );
}
```

The copy constructor checks if the initialization object has data; if not, it simply sets the member variables to their default values. If the object contains data, then the copy constructor allocates a new array, sets the member variables to the values of the initialization object, and copies the data from array of the initialization object. For illustrative purposes, this copy constructor *prepends* (as opposed to appends) the string "copy of" to the name of the object. The copy constructor is called twice in the following code sample:

```
void main()
{
    try
    {
        // Create a record called test
        RECORD a( "test", 100, 5.0f );

        // Create a copy of a in record b
        RECORD b( a );

        // Create a copy of b in record c
        RECORD c( b );
```

```
        // Display the names of the records
        cout << "a: " << a.m_name << endl;
        cout << "b: " << b.m_name << endl;
        cout << "c: " << c.m_name << endl;
    }
    catch( const char *e )
    {
        cerr << e << endl;
        return;
    }
}
```

The output of the program is

```
a: test
b: copy of test
c: copy of copy of test
```

When a structure is deleted, a corresponding member function named the *destructor* is called. The destructor is the name of the structure preceded by a tilde (~). Any memory allocated in a structure can be deleted by the destructor if it wasn't deleted already by a member function. This prevents hard-to-find memory leaks from structures. An example destructor for the **RECORD** structure deletes any memory that may have been allocated during the life of the object:

```
// Destructor
~RECORD()
{
    delete [] m_data;
}
```

which will delete any memory allocated for the data. If memory for data was never allocated during the lifetime of the **RECORD** structure, the **delete** operator will accept **NULL** as an input parameter and return immediately. Destructors do not take parameters nor return any values.

2.8.4 Pointers to Structures

Pointers to structures can be used to dynamically allocate arrays of structures and efficiently access structures within functions. The following code segment can be used to dynamically allocate a five-element array of **RECORD** structures:

```
RECORD *voices = new RECORD[5];
```

This statement is equivalent to the single-array definition **RECORD voices[5];** except that the memory block allocated by **new** can be deallocated by the **delete** operator. With either definition, memory is allocated for the size of the structure multiplied by 5 (the size of the array) and five constructors are called. The length of each element of the array could be printed as follows:

```
for( int i = 0; i < 5; i++ )
    cout << "Length " << i << ": " << voices[i].m_length << endl;
```

The **voices** array can also be accessed by using a pointer to the array of structures. If **ptrvoice** is a **RECORD** pointer (by declaring it with **RECORD *ptrvoice;**), then (***ptrvoice).m_length** could be used to give the length of the **RECORD** that was pointed to by **ptrvoice**. Because this form of pointer operation occurs with structures often, a special operator (**->**) was defined. Thus, **ptrvoice->m_length** is equivalent to **(*ptrvoice).m_length**. This shorthand is very useful when used with functions, since a local copy of a structure pointer is passed to the function. For example, the following function will display the name and length of each record in an array of **RECORD** structures of length **size**:

```
void printRecords( RECORD *ptrvoice, int size )
{
    if( ptrvoice == NULL )
        throw "NULL pointer to array of records";

    for( int i = 0; i < size; i++ )
    {
        cout << "Record: " << ptrvoice->m_name << endl;
        cout << "Length: " << ptrvoice->m_length << endl << endl;
        ptrvoice++;
    }
}
```

Thus, a statement like **printRecords(voices, 5);** will print the names and lengths stored in each of the five elements of the array of **RECORD** structures.

2.9 CLASSES

Classes are exactly like structures except that the default behavior is that no routine outside of the class can access any members, whereas in structures all of the members are accessible by external routines. The concepts of data hiding and encapsulation require the ability to prevent external sources from modifying the internal state of objects. The previous discussion on structures applies to classes as well, but the topics in this section are more commonly seen with classes.

The **RECORD** structure defined above could be converted to a class by changing the **struct** keyword to **class** as follows:

```
class RECORD
{
    // Class declaration
    ...
};
```

2.9.1 Member Access Identifiers

Although the default behavior of classes is to hide all members, there must be a way to access some of the functionality of classes or they would be useless. C++ offers fine-grain control of access to members of structures and classes. The three levels of access control are defined by the keywords **private**, **protected**, and **public**. Declaring members **private** prevents access to them by any routine outside of the class. This is the default behavior for classes. Declaring members **protected** allows derived classes to access data. Class derivation and inheritance is discussed later in Section 2.9.3. Members declared **public** can be accessed by any routine. This is the default behavior for structures. Usually, the data members are declared **private** or **protected** and the functions that external routines call are declared **public**. The **RECORD** class would become

```
class RECORD
{
// Members accessed by this class
private:
    // Member variables
    int m_length;
    char m_name[80];
    int *m_data;

// Members which are accessed by external routines
public:
    // Default Constructor
    RECORD()
    {
        setName( "Unknown data" );
        m_length = 0;
        m_data = NULL;
    }

    // Constructor
    RECORD( const char *name, int length )
    {
        m_data = new int[length];
```

```
            if( m_data == NULL )
                throw "Cannot allocate memory";

            m_length = length;
            setName( name );
        }

        // Destructor
        ~RECORD()
        {
            delete [] m_data;
        }

        // Member functions
        void setName( const char *name )
        {
            if( name == NULL )
                m_name[0] = 0;
            else
                strcpy( m_name, name );
        }
    };
```

Although this class declaration will compile, if we tried to print out the class information using the **printRecords** function defined above, we would receive access violation errors from the compiler. The data members **m_name** and **m_length** are declared private and cannot be accessed by an external routine such as **printRecords**. The solution is to provide a member function that prints the member data of the class. The public member function **printRecordData** is defined inside of the class definition as follows:

```
void printRecordData()
{
    cout << "Record Name: " << m_name << endl;
    cout << "Length: " << m_length << endl;
}
```

Now the routine **printRecords** can iterate through each class in the input parameter **ptrvoice** and call the **printRecordData** member function:

```
void printRecords( RECORD *ptrvoice, int size )
{
    if( ptrvoice == NULL )
        throw "NULL pointer to array of records";

    for( int i = 0; i < size; i++ )
```

```
    {
        ptrvoice->printRecordData();
        ptrvoice++;
    }
}
```

Adding public member functions allows external routines to operate on the data of the class without compromising the integrity of the data.

2.9.2 Operator Overloading

One of the additions to C++ was operator overloading, or the ability for programs to enhance standard operators to work with user-defined types. Routines external to the **RECORD** class cannot access the member variable **m_data,** because it is private. A natural extension to the **RECORD** class would be to allow external routines to access the data like an array using array brackets **[]** but to include validation checking to ensure that the index is within the array boundary. To do this, an operator of type **[]** is declared that takes a single **int** parameter as an index into the data:

```
// Return an element of the data
int& operator[]( int index )
{
    // Parameter validation
    if( m_data == NULL )
        throw "Record has no data";

    if( index < 0 || index >= m_length )
        throw "Record data index out of bounds";

    return m_data[index];
}
```

External routines can access any element of the record data, and the **[]** operator guarantees that there will never be a memory access violation. An example of filling the elements of data from 0 to 99 would be

```
// Create a record
RECORD a( "test", 100 );

// Fill data
for( int i = 0; i < 100; i++ )
    a[i] = i;
```

This operator works on both the left- and right-hand side of the equation. To set the variable **temp** to the 50th element and display the first 10 elements of data would be

```
// Retrieve data
int temp = a[49];
for( i = 0; i < 10; i++ )
    cout << a[i] << " ";
cout << endl;
```

A common operator to overload is the assignment operator (**=**), which sets the contents of one object equal to the contents of another object. The assignment operator takes a reference to the object on the right-hand side as an argument and returns a reference to the result. The assignment operator is similar to the copy constructor mentioned in Section 2.8.3, but with a subtle difference: The copy constructor works on objects when they are first created, and the assignment operator works on objects already created. Because of this difference, the assignment operator must take into account the case where an object is set to itself and the case where there is already data allocated inside of the object. In the sample below, the assignment operator for the **RECORD** class checks if the class it is assigned to is itself. If the assignment is to itself, it simply returns itself with no changes. If a new object is required and if there is any data already allocated, the previous data must be deleted before copying the data from the object on the right-hand side.

```
// Assignment operator
RECORD& operator=( const RECORD& rec )
{
    // Check if record passed in equal to this (a = a)
    if( &rec != this )
    {
        // Different object -- Delete data if allocated
        delete [] m_data;
        m_data = NULL;
        m_length = 0;

        // Check if record has data
        if( rec.m_data != NULL )
        {
            // Allocate space for new data
            m_data = new int[rec.m_length];
            if( m_data == NULL )
                throw "Out of memory";
            // Copy data
            m_length = rec.m_length;
            for( int i = 0; i < m_length; i++ )
                m_data[i] = rec.m_data[i];
        }
    }
    return *this;
}
```

When the assignment operator checks if the class passed in is the same this class with **if(&rec != this)**, it uses the *this* pointer. The this pointer is a keyword in C++ that stands for the address of the current class. All member functions have access to the this pointer and all accesses to member variables implicitly use the this pointer. The **printRecordData** function could have been written using the this pointer explicitly as

```
void printRecordData()
{
    cout << "Record Name: " << this->m_name << endl;
    cout << "Length: " << this->m_length << endl;
}
```

The **this->** is usually omitted because it is understood that variables accessed without **this->** are members of the operator's class. The entire **RECORD** class is shown in Listing 2.2.

```
class RECORD
{
// Members accessed by this class
private:
    //
    // Member variables
    //
    int m_length;
    char m_name[80];
    int *m_data;
// Members which are accessed by external routines
public:
    // Default Constructor
    RECORD()
    {
        setName( "Unknown data" );
        m_length = 0;
        m_data = NULL;
    }

    // Constructor
    RECORD( const char *name, int length )
    {
        m_data = new int[length];
        if( m_data == NULL )
            throw "Cannot allocate memory";

        m_length = length;
```

LISTING 2.2 The RECORD class. (*Continued*)

```
        setName( name );
    }

    // Copy constructor
    RECORD( const RECORD& rec )
    {
        if( rec.m_data == NULL )
        {
            // No data
            m_data = NULL;
            m_length = 0;
        }
        else
        {
            // Data exists -- Create copy of rec
            m_data = new int[rec.m_length];
            if( m_data == NULL )
                throw "Cannot allocate memory";
            m_length = rec.m_length;
            for( int i = 0; i < m_length; i++ )
                m_data[i] = rec.m_data[i];
        }

        // Prepend "copy of" to beginning of name
        strcpy( m_name, "copy of " );
        strcat( m_name, rec.m_name );
    }

    // Destructor
    ~RECORD()
    {
        delete [] m_data;
    }

    //
    // Member functions
    //
    // Set the name of this class
    void setName( const char *name )
    {
        if( name == NULL )
            m_name[0] = 0;
        else
            strcpy( m_name, name );
    }
```

LISTING 2.2 (*Continued*)

```
    // Display class information
    void printRecordData()
    {
        cout << "Record Name: " << m_name << endl;
        cout << "Length: " << m_length << endl;
    }
    //
    // Operators
    //
    // Assignment operator
    RECORD& operator=( const RECORD& rec )
    {
        // Check if record passed in equal to this (a = a)
        if( &rec != this )
        {
            // Different object -- Delete data if allocated
            delete [] m_data;
            m_data = NULL;
            m_length = 0;

            // Check if record has data
            if( rec.m_data != NULL )
            {
                // Allocate space for new data
                m_data = new int[rec.m_length];
                if( m_data == NULL )
                    throw "Out of memory";

                // Copy data
                m_length = rec.m_length;
                for( int i = 0; i < m_length; i++ )
                    m_data[i] = rec.m_data[i];
            }
        }
        return *this;
    }

    // Array index operator
    int& operator[]( int index )
    {
        // Parameter validation
        if( m_data == NULL )
            throw "Record has no data";
        if( index < 0 || index >= m_length )
```

LISTING 2.2 (*Continued*)

```
            throw "Record data index out of boundary";
        return m_data[index];
    }
};
```

LISTING 2.2 (*Continued*)

2.9.3 Inheritance

Inheritance is a mechanism to allow code reuse and is the cornerstone of object-oriented programming. After the **RECORD** class was implemented, additional functionality may be required in different modules of the program. With the C programming language, the **RECORD** structure could be copied into the other module, renamed, and the member variables added. With Inheritance, however, additional functionality can be added to existing classes by *deriving* subclasses from them. Classes derived from **RECORD** are subclasses of record, while **RECORD** is the *base class*. The advantages of inheritance are

1. Code reuse by creating a hierarchy of objects with different functionality
2. Increased robustness by basing new functionality on previously tested modules
3. The ability to combine at run time different object with the same base class together

The **DSPRECORD** class derives from the **RECORD** class and adds the member variable **m_sampleRate** to store additional information. The sample rate is set in the constructor for the **DSPRECORD**. The declaration of **DSPRECORD** becomes:

```
class DSPRECORD : public RECORD
{
// Members accessed by this class
private:
    //
    // Member variables
    //
    float m_sampleRate;

// Members which are accessed by external routines
public:
    // Default Constructor
    DSPRECORD() :
        RECORD(),
        m_sampleRate( 0.0f )
    { }

    // Constructor
    DSPRECORD( const char *name, int length, float sampleRate ) :
        RECORD( name, length ),
```

```
        m_sampleRate( sampleRate )
    { }
};
```

There are two new concepts introduced in this example. First, the syntax to derive the **DSPRECORD** class from the **RECORD** class is to declare the **DSPRECORD** class followed by a colon (**:**), the keyword **public**, then the class name **RECORD**. The second concept is initializing the member variables in the *member initialization list* instead of the body of the constructor. The member initialization list starts after the colon following the constructor declaration. The variables inside of the member initialization list are set just before the class is created. Since the derived class must call the constructor for the base class to initialize **m_length**, **m_name** and **m_data**, the base class must be called from the member initialization list and not the body of the constructor. Other member variables such as **m_sampleRate** can be initialized this way as well.

2.9.4 Complex Numbers

A complex number can be defined by using a structure of two **floats** as follows:

```
struct Complex
{
    float m_real;
    float m_imag;
};
```

A constructor can be defined to take parameters (or assign them to a default value if none are provided) as follows:

```
Complex( float real = 0.0f, float imag = 0.0f ) :
    m_real( real ),
    m_imag( imag ) { }
```

This constructor makes use of default parameters in the declaration and the member initialization list to set **m_real** and **m_imag** to **real** and **imag**, respectively, or to set them to **0.0f** if no parameters are given. Three complex numbers **a**, **b**, and **c** can be defined using the above structure as follows:

```
Complex a( 3.0f, 4.0f ); // m_real = 3.0f, m_imag = 4.0f
Complex b( 5.0f, 6.0f ); // m_real = 5.0f, m_imag = 6.0f
Complex c;               // (default) m_real = 0.0f, m_imag = 0.0f
```

In order to perform the complex addition **c = a + b;**, an addition operator can be defined. Since the structure has public access on all member by default, the operator does not have to be a member of the class:

```
// Add two complex numbers
Complex operator+( const Complex& c1, const Complex& c2 )
{
    Complex ret( c1.m_real + c2.m_real, c1.m_imag + c2.m_imag );
    return ret;
}
```

Now the following statement adds the two complex numbers **a** and **b** and places the result in **c**:

```
c = a + b;
```

If the addition operator was not defined, this statement would have caused a compile-time error because the compiler doesn't know how to add two classes together.

2.10 INPUT AND OUTPUT

Two types of input/output classes will be briefly described in this section. The console input/output classes **cin**, **cout**, and **cerr** will be described in the next section, and the functions required to access disk files will be considered in Section 2.10.2. Further details concerning the standard input/output functions can be found in Appendix A.

2.10.1 `cin`, `cout`, and `cerr`

The three standard classes that allow simple formatted dialogue with the keyboard and display are **cin**, **cout**, and **cerr**. **cin** "scans" a line entered by the user and generates data in a number of variables that are arguments to the overloaded extraction operator (**>>**). **cout** and **cerr** display the results of a program by sending output to the display device via the overloaded insertion operator (**<<**). The difference between **cout** and **cerr** is that **cout** writes to standard output stream and **cerr** writes to the standard error stream. Both will usually appear on the screen, but command-line arguments can redirect one or both streams to different devices or files. All classes can take standard types as parameters.

The following code requests an integer, floating-point value, and character string from the user and then displays them:

```
int value;
float scale;
char name[80];

cout << "Enter value: ";
cin >> value;
cout << "Enter scale: ";
cin >> scale;
```

```
cout << "Enter name: ";
cin >> name;

cout << "Value: " << value << endl;
cout << "Scale: " << scale << endl;
cout << "Name: " << name << endl;
```

The insertion and extraction operators can be overloaded for user-defined types such as the **Complex** class described in Section 2.9.4:

```
ostream &operator<<( ostream& os, const Complex& c )
{
    os << "( " << c.m_real << ", " << c.m_imag << " )";
    return os;
}

istream &operator>>( istream& is, Complex& c )
{
    is >> c.m_real >> c.m_imag;
    return is;
}
```

Now that the stream operators are overloaded, when the programmer wants to display a complex number or read a pair of numbers from the user, the **cout** and **cin** operators can be used as follows:

```
Complex a, b;
cout << "Enter complex number A: ";
cin >> a;
cout << "Enter complex number B: ";
cin >> b;
cout << "A + B = " << ( a + b ) << endl;
```

2.10.2 Accessing Disk Files

The **fstream** class supports disk file input and output. Disk files can be accessed using the standard classes in the following ways:

1. Character by character using **get()** and **put()**
2. As a random access binary file using **read(), write(),** and **seek()**
3. With formatted output via the insertion and extraction operators (**<<**) and (**>>**)

The exact format of each of the functions listed above can be found in Appendix A or the manual for the compiler to be used. In both access methods, the file must first be opened using the **open** member function. The **open** member function has the following format:

```
void fstream::open( const char* fileName, int mode, int prot );
```

where **fileName** is a null-terminated string that gives the file name that will be passed to the operating system. The mode specifies the type of open to perform and is one or more of the following values:

ios::app	Open and append to end of file.
ios::in	Open file for input.
ios::out	Open file for output.
ios::trunc	Open and truncate file.
ios::nocreate	Open existing file.
ios::noreplace	Open if file doesn't exist.
ios::binary	Opens the file in binary mode.

The protection mode, specified by **prot**, tells the operating system if other programs can access the file while it is open. The **fstream** class and related constants are in the standard header files **iostream.h** and **fstream.h.** The header files must be included (using the preprocessor **#include** directive) at the beginning of any program that makes use of files. If the file cannot be opened when **fstream::open** is invoked, the fail member function in the fstream class will return nonzero and can then be tested by the calling program as illustrated by the example program shown in Listing 2.3.

A file is closed by calling the **close** member function of the **fstream** class. A file must be closed before a program is terminated so that any buffers that may have been established (especially in text modes) can be flushed out and so that the operating system may update the directory entry.

Listing 2.3 shows a short program that reads a binary file of integers and creates a binary file of 32-bit floats to illustrate binary file input and output. A file **(INT.DAT)** consisting of a series of 16-bit ints is opened for binary read. A new file **(FLOAT.DAT)** is created, and a floating-point number is written to it for each integer found in the **INT.DAT** file.

2.11 COMMON C++ PROGRAMMING PITFALLS

The following sections describe some of the more common errors made by programmers when they first start coding in C++ and a few suggestions concerning how to avoid them.

2.11.1 Special String Characters

In a program, a string constant is a set of characters enclosed by a pair of double quotes (e.g., "string"). There are several special character sequences (called *escape sequences*) used to delineate characters that are not normally printed. All of these special characters

```
#include <iostream.h>
#include <fstream.h>

void main()
{
    try
    {
        // Two file classes
        fstream fsIn, fsOut;

            fsIn.open( "int.dat", ios::in | ios::nocreate | ios:: binary );
        if( fsIn.fail() )
            throw “Error opening int.dat”;

            fsOut.open( "float.dat", ios::out | ios::trunc | ios::binary );
        if( fsOut.fail() )
            throw "Error opening float.dat";

        // Do until break by EOF or error
        while( true )
        {
            short int read = 0;
            float written = 0.0f;

            // Read 16-bit integer from file
            fsIn.read( (char *)&read, sizeof( read ) );
            if( fsIn.eof() )
                break;
            if( fsIn.fail() )
                throw "Input failed";

            // Convert to floating-point value
            written = (float)read;

            // Write floating-point value to file
            fsOut.write( (char *)&written, sizeof( written ) );
            if( fsOut.fail() )
                throw “Output failed”;
        }
    }
    catch( const char *e )
    {
        cerr << e << endl;
    }
}
```

LISTING 2.3 Binary file input and output example program.

start with a backslash (\) and are listed in Section 2.2.3. The next character (or characters) tells the compiler what kind of character to insert in the string (e.g., **\r** inserts a carriage return in the string). These special character strings are very useful but can sometimes cause problems. One example of misuse of special characters in strings that occurs frequently when using the MS-DOS operating system (and derivatives that maintain file compatibility) is as follows:

```
fs.open("c:\tom\new.dat", ios::in );
```

This statement was intended to open the file **new.dat** in the subdirectory called **tom**. Because the MS-DOS file system indicates subdirectories with the backslash (\) and C++ uses the backslash for escape sequences, this statement will never open the file. To see what happened we could execute the statement

```
cout << "c:\tom\new.dat";
```

which would write "c:", give a horizontal tab (because of the **\t**) followed by the characters **om**, then a newline (because of the **\n**) followed by **sp.dat,** which will never be a valid file name. The correct statement to open the file is simply

```
fs.open( "c:\\tom\\new.dat", ios::in );
```

The **** is the escape sequence for ****. When placing fixed file names in a program, it is best not to use file strings with subdirectory names (or *paths,* as they are called) because it may make the program hard to use on other computers that do not have a particular subdirectory and also because the program will not be able to run under other operating systems that do not use \ to indicate subdirectories. Another common escape sequence is \" for the double quote ("). This is required because a double quote in the middle of a string would probably result in a compiler error (a premature end of the string).

2.11.2 Array Indexing

All array indices start with zero rather than one. This makes the last index of an N long array N-1. This is very useful in digital signal processing because many of the expressions for filters, z-transforms, and FFTs are easier to understand and use with the index starting at zero instead of one. For example, the FFT output for $k = 0$ gives the zero frequency (DC) spectral component of a discrete time signal. A typical indexing problem is illustrated in the following code segment that is intended to determine the first 10 powers of 2 and store the results in an array called **power2**:

```
int power2[10];
int p = 1;
for( int i = 1; i <= 10; i++ )
{
```

```
        power2[i] = p;
        p = 2 * p;
    }
```

This code segment will compile well and may even run without any difficulty. The problem is that the **for** loop index **i** stops on **i = 10** and **power2[10]** is not a valid index to the **power2** array. Also, the **for** loop starts with the index 1 causing **power2[0]** not to be initialized. This results in the first power of two (2^0, which should be stored in **power2[0]**) to be placed in **power2[1]**. One way to correct this code is to change the for loop to read **for(int i = 0; i < 10; i++)** so that the index to **power2** starts at 0 and stops at 9.

2.11.3 Misusing Pointers

Since pointers are new to many programmers, the misuse of pointers can be particularly difficult because most compilers will not indicate any pointer errors (some compilers issue a warning for some pointer errors). Some pointer errors will result in the program not working correctly, or, worse yet, the program may seem to work but will not work with a certain type of data or when the program is in a certain mode of operation. On many small single-user systems, misused pointers can easily result in writing to memory that is used by the operating system, often resulting in a system "crash" and requiring a subsequent reboot.

There are two types of pointer abuses: setting a pointer to the wrong value (or not initializing it at all) and confusing arrays with pointers. The following program shows both of these problems:

```
#include <iostream.h>

void main()
{
    char *string;
    char msg[10];
    cout << "Enter title\n";
    cin.getline( string, 80 );
    for( int i = 0; *string != ' '; i++ )
        string++;

    msg = "\n\nTitle = ";
    cout << msg << string << endl;
    cout << i << " characters before first space\n";
}
```

After including the required header file to display text and receive user input, the first two statements in the main routine body declare that memory be allocated to a pointer variable called **string** and a 10-element **char** array called **msg**. Next the user is asked to enter

a title into the variable called **string**. The **for** loop is intended to search for the first space in the string and the last statement is intended to display the string after the first space.

There are three pointer problems in this program although the program will compile with only one fatal error (and a possible warning). The fatal error message will reference the **msg = "\n\nTitle =";** statement. This line tells the compiler to set the address of the **msg** array to the constant string **"\n\nTitle ="**. This is not allowed so the error "Lvalue required" (or something less useful) will be produced. The roles of an array and a pointer have been confused and the **msg** variable should have been declared as a pointer and used to point to the constant string **"\n\nTitle =",** which was already allocated storage by the compiler.

The next problem with the code segment is that **getline** will read the string into the address specified by the argument **string**. Unfortunately, the value of **string** at execution time could be anything (some compilers will set it to zero) that will probably not point to a place where the title string could be stored. Some compilers will issue a warning indicating that the pointer called string may have been used before it was defined. The problem can be solved by initializing the string pointer to a memory area that is allocated for storing the title string. The memory can be dynamically allocated by a simple call to **new** as shown in the following improved program:

```
#include <iostream.h>

void main()
{
    char *string = new char[80];
    if( string == NULL )
        throw "Out of memory";

    cout << "Enter title\n";
    cin.getline( string, 80 );
    for( int i = 0; *string != ' '; i++ )
        string++;

    char *msg = "\n\nTitle = ";
    cout << msg << string << endl;
    cout << i << " characters before first space\n";
}
```

The code will now compile and run but will not give the correct response when a title string is entered. In fact, the first characters of the title string before the first space will not be printed, because the pointer **string** was moved by the execution of the **for** loop. This may be useful for finding the first space in the **for** loop but results in the address of the beginning of the string being lost. It is best not to change a pointer that points to a dynamically allocated section of memory. Also, since the memory was dynamically allocated with the **new** operator, it was never released before the end of the program, and possible memory leaks could occur. Both of these pointer problems can be fixed by using

a temporary pointer (the **char** pointer variable called **cp**) for the **for** loop and deleting the **string** variable at the end of the program as follows:

```
#include <iostream.h>

void main()
{
    char *string = new char[80];
    if( string == NULL )
        throw "Out of memory";

    cout << "Enter title\n";
    cin.getline( string, 80 );
    char *cp = string;
    for( int i = 0; *cp != ' '; i++ )
        cp++;

    char *msg = "\n\nTitle = ";
    cout << msg << string << endl;
    cout << i << " characters before first space\n";
    delete [] string;
}
```

Another problem with this program segment is that if the string entered contains no spaces, then the **for** loop will continue to search through memory until it finds a space. The program will almost always find a space (in the operating system perhaps) and will set **i** to some large value. On larger multi-user systems this may result in a fatal run-time error because the operating system must protect memory not allocated to the program. Although this programming problem is not unique to C or C++, it does illustrate an important characteristic of pointers—pointers can and will point to any memory location without regard to what may be stored there. The solution is to check for the terminating null character in the **for** loop expression and break out of the loop if a space is encountered as follows:

```
#include <iostream.h>

void main()
{
    char *string = new char[80];
    if( string == NULL )
        throw "Out of memory";

    cout << "Enter title\n";
    cin.getline( string, 80 );
    char *cp = string;
    for( int i = 0; *cp != 0; i++ )
```

```
    {
        if( *cp == ' ' )
            break;
        cp++;
    }

    char *msg = "\n\nTitle = ";
    cout << msg << string << endl;
    if( *cp == 0 )
        cout << "No spaces in string\n";
    else
        cout << i << " characters before first space\n";
    delete [] string;
}
```

2.12 COMMENTS ON PROGRAMMING STYLE

The four common measures of good DSP software are *reliability, maintainability, extensibility,* and *efficiency*.

A reliable program is one that seldom (if ever) fails. This is especially important in DSP because tremendous amounts of data are often processed using the same program, and if the program fails due to one sequence of data passing through the program, it may be difficult, or impossible, to ever determine what caused the problem.

Since most programs of any size will occasionally fail, a maintainable program is one that is easy to fix. A truly maintainable program is one that can be fixed by someone other than the original programmer. It is also sometimes important to be able to maintain a program on more than one type of processor, which means that in order for a program to be truly maintainable it must be portable.

An extensible program is one that can be easily modified when the requirements change, new functions need to be added, or new hardware features need to be exploited.

An efficient program is often the key to a successful DSP implementation of a desired function. An efficient DSP program will use the processing capabilities of the target computer (whether general purpose or dedicated) in such a way as to minimize the execution time. In a typical DSP system, this often means minimizing the number of operations per input sample or maximizing the number of operations that can be performed in parallel. In either case, minimizing the number of operations per second usually means a lower overall system cost, as fast computers typically cost more than slow computers. For example, it could be said that the FFT algorithm reduced the cost of speech processing (both implementation cost and development cost) such that inexpensive speech recognition and generation processors are now available for use by the general public.

Unfortunately, DSP programs often forsake maintainability and extensibility for efficiency. Such is the case for most currently available programmable signal processing integrated circuits. These devices are usually programmed in assembly language in such a way that it is often impossible for changes to be made by anyone but the original pro-

grammer (and after a few months even the original programmer may have to rewrite the program to add additional functions). Often a compiler is not available for the processor, or the processor's architecture is not well suited to efficient generation of code from a compiled language. Fortunately, the current trend in programmable signal processors appears to be toward high-level languages such as C and C++.

2.12.1 Software Quality

The four measures of software quality (reliability, maintainability, extensibility, and efficiency) are rather difficult to quantify. One almost has to try to modify a program to find out if it is maintainable or extensible. A program is usually tested in a finite number of ways much smaller than the millions of input data conditions. This means that a program can be considered reliable only after years of bug-free use in many different environments.

Programs do not acquire these qualities by accident. It is unlikely that good programs will be intuitively created just because the programmer is clever, experienced, or uses lots of comments. Even the use of structured programming techniques (described briefly in the next section) will not assure that a program is easier to maintain or extend. It is the authors' experience that the use of the following in coding will often lessen the software quality of DSP programs:

1. Functions that are too big or have several purposes.
2. A main program that does not use functions.
3. Functions that are tightly bound to the main program.
4. Programming "tricks" that are always poorly documented.
5. Lack of meaningful variable names and comments.

An *oversized function* (item 1) might be defined as one that exceeds two pages of source listing. A function with more than one purpose lacks strength. A function with one clearly defined purpose can be used by other programs and other programmers. Functions with many purposes will find limited utility and limited acceptance by others. All of the functions described in this book and contained on the included disk were designed with this important consideration in mind. Functions that have only one purpose should rarely exceed one page. This is not to say that all functions will be smaller than this. In time-critical DSP applications the use of "in-line" code can easily make a function quite long but can sometimes save precious execution time. It is generally true, however, that big programs are more difficult to understand and maintain than small ones.

A *main program* that does not use functions (item 2) will often result in an extremely long and hard-to-understand program. Also, because complicated operations often can be independently tested when placed in short functions, the program may be easier to debug. However, if this rule is taken to the extreme it can result in functions that are *tightly bound* to the main program violating rule 3. A function that is tightly bound to the rest of the program (by using to many global variables, for example) weakens the en-

tire program. If there are lots of tightly coupled functions in a program, maintenance becomes impossible. A change in one function can cause an undesired, unexpected change in the rest of the functions.

Clever programming tricks (item 4) should be avoided because they will often not be reliable and will almost always be difficult for someone else to understand (even with lots of comments). Usually if the program timing is so close that a "trick" must be used, then the wrong processor was chosen for the application. Even if the programming trick solves a particular timing problem, as soon as the system requirements change (as they almost always do), a new timing problem without a solution may soon develop.

A program that does not use *meaningful variables and comments* (item 5) is guaranteed to be very difficult to maintain. Consider the following valid program:

```
main(){int _o_oo_,_ooo;for(_o_oo__=2;;__o__o__++)
{for(__ooo_=2;_o__oo__%__ooo_!=0;__ooo_++;
if(__ooo_==_o_oo__)cout<<_o_oo__<< endl;}}
```

Even the most experienced programmer would have difficulty determining what this three-line program does. Even after running such a poorly documented program, it may be hard to determine how the results were obtained. The following program does exactly the same operations as the above three lines but is easy to follow and modify:

```
main()
{
    // The outer for loop trys all numbers > = 2 and the inner
    // for loop (with no statements in the loop) checks the number
    // tested (primeTest) for any divisors less than itself. If
    // divisor is equal to primeTest the first divisor found was
    // primeTest so it is printed as a prime number.
    for( int primeTest = 2;; primeTest++ )
    {
        for( int divisor = 2; ( primeTest % divisor ) != 0; divisor++ )
            ;

        if( divisor == primeTest )
            cout << primeTest << endl;
    }
}
```

It is easy for anyone to discover that the above well-documented program prints a list of prime numbers because the following three documentation rules were followed:

1. Variable names that are meaningful in the context of the program are used. Avoid variable names such as **x**,**y**,**z** or **i**,**j**,**k** unless they are simple indexes used in a very obvious way (such as initializing an entire array to a constant).

2. Comments precede each major section of the program (the above program only has one section). Although the meaning of this short program is fairly clear without the comments, it rarely hurts to have too many comments. Adding a blank line between different parts of a program also improves the readability of a program because the different sections of code appear separated from each other.
3. Statements at different levels of "nesting" are indented to show which control structure controls the execution of the statements at a particular level.

2.12.2 Structured Programming

Structured programming has developed from the notion that any algorithm, no matter how complex, can be expressed by using the programming control structures *if-else, while,* and *sequence.* All programming languages must contain some representation of these three fundamental control structures. The development of structured programming revealed that if a program uses these three control structures, then the logic of the program can be read and understood by beginning at the first statement and continuing downward to the last. Also, all programs could be written without **goto** statements. Generally, structured programming practices lead to code that is easier to read, easier to maintain, and even easier to write.

There are the three basic control structures as well as three additional structured programming constructs called *do-while, for,* and *case.* The additional three control structures have been added to most modern languages because they are convenient, they retain the original goals of structured programming, and their use often makes a program easier to comprehend.

The *sequence* control structure is used for operations that will be executed once in a function or program in a fixed sequence. This structure is often used where speed is most important and is often referred to as in-line code when the sequence of operations is identical and could be coded using one of the other structures. Extensive use of in-line code can obscure the purpose of the code segment.

The *if-else* control structure is the most common way of providing conditional execution of a sequence of operations based on the result of a logical operation. Indenting of different levels of **if** and **else** statements (as shown in the example in Section 2.4.1) is not required; they are an expression of programming style that helps the readability of the if-else control structure. Nested **while** and **for** loops should also be indented for improved readability (as was illustrated in Section 2.7.3).

The *case* control structure is a convenient way to execute one of a series of operations based upon the value of an expression (see the example in Section 2.4.2). It is often used instead of a series of if-else structures when a large number of conditions are tested based upon a common expression. The **switch** statement gives the expression to test, and a series of **case** statements gives the conditions to match the expression. A **default** statement can be optionally added to execute a sequence of operations if none of the listed conditions is met.

The last three control structures (*while, do-while,* and *for*) all provide for repeating a sequence of operations a fixed or variable number of times. These loop statements can make a program easy to read and maintain. The **while** loop provides for the iterative execution of a series of statements as long as a tested condition is true; when the condition is false, execution continues on to the next statement in the program. The **do-while** control structure is similar to the **while** loop except that the sequence of statements is executed at least once. The *for* control structure provides for the iteration of statements with automatic modification of a variable and a condition that terminates the iterations. **For** loops allow for any initializing statement, any iterating statement, and any terminating statement. The following three examples of a **while**, **do-while**, and **for** loop all calculate the power of two of an integer i (assumed to be greater than 0) and set the result to **k:**

```
int k = 2;          // while loop k=2**i
while( i > 0 )
{
    k = 2 * k;
    i—;
}

k = 1;              // do-while loop k = 2**i
do
{
    k = 2 * k;
    i—;
}
while( i > 0 );

for( k = 2; i > 1; i— ) // for loop k=2**i
    k = 2 * k;
```

The preference as to which form of loop to use is a personal matter. Of the three equivalent code segments shown above, the **for** loop and the **while** loop both seem easy to understand and would probably be preferred over the **do-while** construction.

There are also several extensions to the six structured programming control structures. Among these are **break**, **continue**, and **goto** (see Section 2.4.5). **Break** and **continue** statements allow the orderly interruption of events that are executing inside of loops. They can often make a complicated loop very difficult to follow, since more than one condition may cause the iterations to stop. The infamous **goto** statement is also included. Nearly every language designer includes a **goto** statement with the advice that it should never be used along with an example of where it might be useful.

The program examples in the chapters that follow and the programs contained on the enclosed disk were developed by using structured-programming practices. The code can be read from top to bottom, there are no **goto** statements, and the six accepted control structures are used. One requirement of structured programming that was not adopted

throughout the software in this book is that each program and function have only one entry and exit point. Although every function and program do have only one entry point (as is required), many of the programs and functions have multiple exit points. Typically, this is done in order to improve the readability of the program. For example, error conditions in a main program often require terminating the program prematurely after displaying an error message. Such an error-related exit is performed by throwing the suitable exception and catching it in an exception handler, if desired. Similarly, some functions have more than one **return** statement, as this can make the logic in a function much easier to program and in some cases more efficient.

2.13 REFERENCES

CARGILL, T., *C++ Programming Style*, Addison-Wesley, Reading, MA, 1992.

ELLIS, M., and STROUSTRUP, B., *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990.

FEUER, A. R., *The C Puzzle Book*, Prentice Hall, Englewood Cliffs, NJ, 1982.

KERNIGHAN, B., and PLAUGER, P., *The Elements of Programming Style*, McGraw-Hill, New York, 1978.

KERNIGHAN, B. W., and RITCHIE, D. M., *The C Programming Language,* 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1988.

MEYERS, S., *Effective C++*, Addison-Wesley, Reading, MA, 1992.

MEYERS, S., *More Effective C++*, Addison-Wesley, Reading, MA, 1996.

PRATA, S., *Advanced C Primer++*, Howard W. Sams and Company, Indianapolis, IN, 1986.

PURDUM, J., and LESLIE, T. C., *C Standard Library*, Que Co., Indianapolis, IN, 1987.

ROCHKIND, M. J., *Advanced C Programming for Displays*, Prentice Hall, Englewood Cliffs, NJ, 1988.

STEVENS, A., *C Development Tools for the IBM PC*, Prentice Hall, Englewood Cliffs, NJ, 1986.

STROUSTRUP, B., T*he C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.

WAITE, M., PRATA, S., and MARTIN, D., *C Primer Plus*, Howard W. Sams and Co., Indianapolis, IN, 1987.

WEINER, R. S., and PINSON, L. J., *An Introduction to Object-Oriented Programming and C++*, Addison-Wesley, Reading, MA, 1988.

CHAPTER 3

User Interface and Disk Storage Routines

Two topics related specifically to C++ programming are discussed in this chapter. When developing DSP functions, the programmer usually writes a test program that exercises the function with a small set of input data. This test program requires some type of simple user interface so that parameters can be entered and the output can be verified. When the function is adequately tested (it is often difficult to know how much testing is adequate), the function is ready to be used in a DSP application program. If the application is a DSP simulation of real-time hardware, then a simple user interface is often all that is required. In both of these frequently encountered DSP development stages, the user interface functions presented in Section 3.1 can be used effectively. If a DSP function is intended for a real-time DSP processor with real-time input and output, other tests using processor-specific simulators, logic analyzers, or even oscilloscopes are often required. Because this type of real-time testing is almost always processor specific, real-time testing is beyond the scope of this book. However, many of the C++ functions presented in subsequent chapters can be applied to real-time systems.

Section 3.2 presents a set of C++ functions that are also useful when testing or simulating DSP algorithms. Digital signal processing systems often require a large amount of data to be processed and generate a large amount of output data (although most DSP systems generate less data than is processed). Disk mass storage is usually required to store the input and output data streams. One major advantage of storing the input digital data on a disk is that the same data can be used over and over again with different algorithms. Thus, the performance of two algorithms can be compared with identical input (provided that the input data is large enough that the comparison can be made fairly). Storing DSP data on disk creates two problems:

1. The input and output data format required by many DSP algorithms is not efficiently supported by most mass storage disk systems.
2. The data on the disk must be identified and organized.

The set of C++ functions and disk data format described in Section 3.2 attempt to provide an efficient yet easy to use method of organizing, storing, and retrieving digital signal processing data.

Section 3.3 presents a program that can be used to display digital signals graphically. This program displays the digital records stored in the disk data format (presented in Section 3.2) as two-dimensional plots of the sample values versus the index of each sample. This program is contained on the accompanying disk and can be used with any IBM-compatible color graphics adapter (CGA) installed in an IBM-PC-compatible computer. Readers who wish to use a more sophisticated graphic display may do so by writing an interface program between their favorite graphics device (or graphics display package) and the disk data format presented in Section 3.2.

3.1 USER INTERFACE

User interface refers to the way that a computer and a user communicate. The most basic user interface on modern computers is the alphanumeric keyboard and display screen. Although the keyboard and display screen are much more interactive user interfaces than the punch cards and line printers of the past, user interfaces continue to improve and change. High-resolution color graphic displays, touch-sensitive screens, trackballs, and mice are examples of popular user interface devices on modern computers and workstations. In fact, user interface techniques form a complete field of study in themselves. Perhaps future user interfaces will include tactile feedback devices and speech recognition. Each new user interface device has its own set of applications and its own set of hardware and software challenges. Because almost all modern computers support the standard alphanumeric keyboard and display (and will continue to do so for some time), this section presents C++ routines that require only these two devices. Strings of characters or numerical values can be read from the user (with convenient prompts and error checking), and output can be generated using standard C++ functions. All of the C++ functions described in this section are contained in the file GET.CPP on the accompanying disk. The functions may be used by including the header file GET.H in the C++ source file and linking with the GET.CPP object (either by specifying the object directly or by using the DSP library—see Appendix B for more information concerning the use of the DSP library and functions).

3.1.1 String Input

Listing 3.1 shows a relatively simple overloaded function called **getInput,** which prompts the user for input and then reads a string from the keyboard. The code shown is fairly self-explanatory and can be used in a program as follows:

```
//////////////////////////////////////////////////////////////
//
// getInput( const char *msg, String &input )
//   Get a string from the user.
//
//  Throws:
//   DSPException
//
//////////////////////////////////////////////////////////////
void getInput( const char *msg, String &input )
{
    const int MAX_LINE_LENGTH = 80;
    char buf[MAX_LINE_LENGTH] = { 0 };

    if( msg != NULL )
       cout << msg << " : ";
    else
       cout << "Enter string: ";

    cin.getline( buf, MAX_LINE_LENGTH );
    if( cin.eof() )
       throw DSPInputException();

    input = buf;
}
```

LISTING 3.1 Function **getInput** (contained in GET.H) used to read a character string from the keyboard with a prompt string.

```
String strName;
// Get parameters from user
do getInput( "Enter your name", strName );
while( strName.isEmpty() );
```

If this code segment is executed in a main program it would result in the following typical dialogue:

Enter your name : **Jane Doe**

In this example, the function **getInput** performed the following steps:

1. An 80-character input buffer was allocated.
2. A prompt string was formed using **<<** and the pointer to the **msg** passed by the caller (in the above example, **"Enter your name"**).
3. Input characters were read from the keyboard until the carriage return and the characters were stored in the allocated input buffer.
4. A pointer to the input string was returned to the caller.

The only restriction on the use of the **getInput** routine is that the input string must be less than 80 characters, otherwise the allocated buffer area will be too small, causing the **getline** routine to overrun the allocated memory. This problem can be fixed by making the input buffer larger than 80 by changing the constant **MAX_LINE_LENGTH**.

3.1.2 Numerical Input

The **getInput** function is also overloaded to read numerical values from the user in floating-point or integer format. The function **getInput** prompts the user for input, checks that the input is in a valid range of values, and then returns a value to the calling program. The function **getInput** is overloaded using **template <class Type>.** Based on **Type** the function will return any valid C++ type of numerical data (integer, single-precision, or double-precision, for example). The examples that follow in this chapter and the rest of the book contain numerous uses of both numerical input routines. Listing 3.2 shows the **getInput** function, which is used to obtain one numeric value from the user. The caller must pass a prompt string and the upper and lower limits of the input value. If the input value entered by the user is above the upper limit or below the lower limit, the user is prompted for input until valid data is entered. A typical use of **getInput** is as follows:

```
int n;
getInput("Enter number of points",n,1,25);
```

When this code is executed (with some bad user input) the following dialogue results:

```
Enter number of points [1..25] : 26

Enter number of points [1..25] : 0

Enter number of points [1..25] : 5text

Enter number of points [1..25] : 10
```

Note that when alphanumeric input was provided or when the numbers were outside the valid range, the user was again prompted until finally the correct value of 10 was entered. The prompt string is created by using the C++ string output operators and the **msg** string passed by the caller. Also, because **getInput** is a template with **class Type** information, any numeric type (or user-defined class) can be used with **getInput**.

3.2 FORMATTED DISK STORAGE

This section presents a set of C++ functions that read and write digital data stored on a disk in a special format suitable for DSP applications. Each function uses the standard C++ binary disk access functions fread and fwrite (see Appendix A). The data is stored

```
//////////////////////////////////////////////////////////////////////
//
// template <class Type>
// getInput(const char *msg, Type &input, Type rangeMin, Type rangeMax )
//   Continue displaying message msg until user
//   gives valid Type input.
//
//  Throws:
//   DSPException
//
//////////////////////////////////////////////////////////////////////
template <class Type>
void getInput( const char *msg, Type &input, Type rangeMin, Type rangeMax )
{
    if( rangeMin > rangeMax )
        throw DSPParamException();

    do
    {
        if( msg != NULL )
            cout << msg << " [" << rangeMin << ".." << rangeMax << "] : ";
        else
           cout << "Enter number [" << rangeMin << ".." << rangeMax << "] : ";

        cin >> input;
        if( cin.eof() )
           throw DSPInputException();
    }
    while( input < rangeMin || input > rangeMax );

    // Read trailing newline
    cin.get();
}
```

LISTING 3.2 Function **getInput** (contained in GET.H) used to read a number from the keyboard with prompt string and range checking.

on disk in a flexible format called the *DSP data format* as illustrated in Figure 3.1. As shown, each DSP data file consists of a fixed-size header, a variable-size data area, and a variable-length descriptive text area called the *trailer.* The *header* gives the type of data stored in the data area, the size of the data elements, and the number of data elements in the data area.

The data area of a DSP data file contains the data as prescribed in the header and is stored in a binary format consistent with the data format used by the machine where the data exists. Thus, floating-point data (declared as a **float**) is usually stored in 4 bytes per data element but the actual assignment of the bits may be machine or even com-

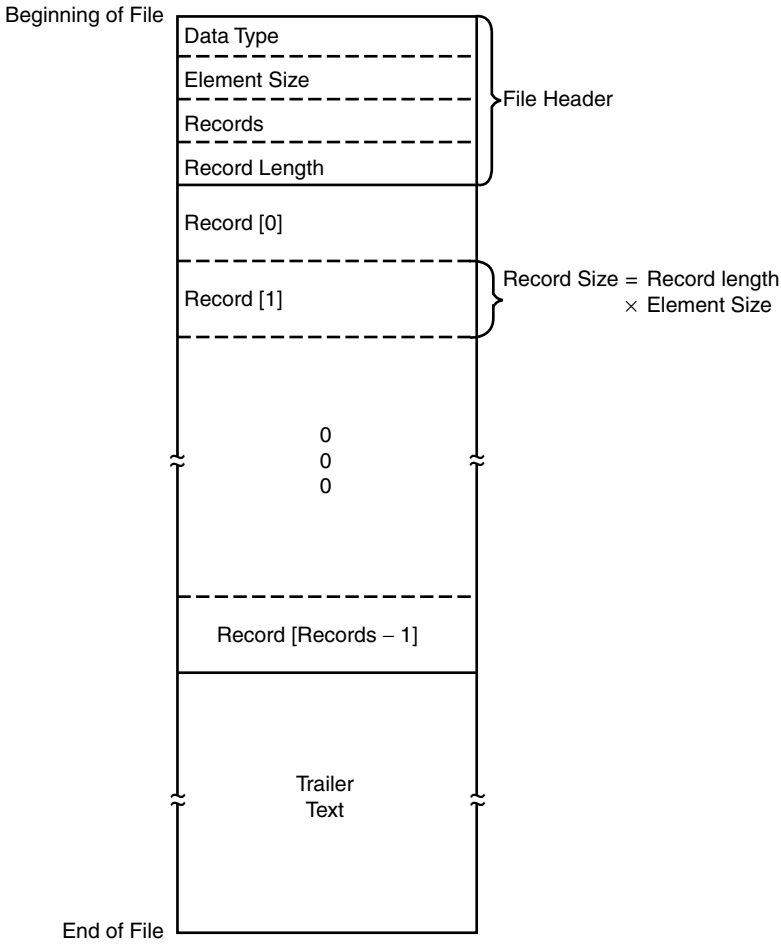


FIGURE 3.1 DSP disk data format consisting of header, data, and trailer areas.

piler specific. Although the integer formats are somewhat more standard (two's complement is most common) data portability problems often exist between different machines (such as the order of the most significant and least significant bytes). Thus, if data portability is important, the user may wish to create a set of C++ functions utilizing a more strict definition of the data format than is implied in the disk data format discussed here.

The trailer section of a DSP data file is an ASCII string that describes the data. The trailer is optional but should be included in order to document the contents of the data file as well as the conditions under which the data was created, the way the data was processed, the time the data was stored, or other information about the data set.

The header is a C++ structure that is written or read directly from disk. The first part of the **DSPFile** class (in file DISK.H) is defined as follows:

```
class DSPFile
{
private:
    // Data type 0-7 as defined above
    DSPFILETYPE m_type;

    // Size of each element
    unsigned char m_elementSize;

    // Number of records
    int m_numRecords;

    // Number of elements in each record
    int m_recLen;

    // Pointer to trailer text
    char *m_trailer;

    // File data
    fstream m_fs;
    bool m_readOnly;
    streampos m_posBeginData;

    // Buffer for reading single elements
    BYTE *m_singleElement;

    ////////////////////////////////////////////////////////////////
    //
    // Helper functions
    //
    ////////////////////////////////////////////////////////////////

    // Clear data
    void empty();
public:

    ////////////////////////////////////////////////////////////////
    //
    // Constructor and Destructor
    //
    ////////////////////////////////////////////////////////////////
    DSPFile();
    ~DSPFile();
```

```
    ////////////////////////////////////////////////////////////
    //
    // File operations
    //
    ////////////////////////////////////////////////////////////
    static bool isFound( const char *name );
    bool hasData() { return( ( m_numRecords * m_recLen ) != 0 ); }
    void openRead( const char *name );
    void openWrite( const char *name );
    void close();

    ////////////////////////////////////////////////////////////
    //
    // Get file format
    //
    ////////////////////////////////////////////////////////////
    DSPFILETYPE getType() { return m_type; }
    int getNumRecords() { return m_numRecords; }
    int getRecLen() { return m_recLen; }

    ////////////////////////////////////////////////////////////
    //
    // Get and Set file contents
    //
    ////////////////////////////////////////////////////////////
    void getTrailer( String& str );
    void setTrailer( const char *trailer );
    void seek( int len = 0, int rec = 0 );
```

The first element in the structure defines the type of data in the data area. Eleven types have been defined and represented by values from 0 to 10. The C++ **const** statement is used in order to conveniently use these types in other C++ functions as follows:

```
typedef unsigned char DSPFILETYPE;
// C DSP_FILE compatible types
const DSPFILETYPE UNSIGNED_CHAR  = 0;
const DSPFILETYPE UNSIGNED_SHORT = 1;
const DSPFILETYPE UNSIGNED_LONG  = 2;
const DSPFILETYPE FLOAT          = 3;
const DSPFILETYPE SIGNED_CHAR    = 4;
const DSPFILETYPE SIGNED_SHORT   = 5;
const DSPFILETYPE SIGNED_LONG    = 6;
const DSPFILETYPE DOUBLE         = 7;

// Defines for addtional data types (C++ DSPFile only)
```

```
const DSPFILETYPE UNSIGNED_INT   = 8;
const DSPFILETYPE SIGNED_INT     = 9;

// Complex numbers
const DSPFILETYPE COMPLEX        = 10;

// Unknown type
const DSPFILETYPE UNKNOWN_TYPE   = 0xFF;
```

The previous definitions are used by the disk class to make the code easier to understand and modify. Each function used to manipulate the disk data format passes the file information via the **DSPFile** class. The first few **DSPFile** member functions (in file DISKIO.CPP) are defined as follows:

```
////////////////////////////////////////////////////////////////////
//
// diskio.cpp - Routines to read and write DSPFiles
//   Contains functions:
//   isFound     Determine if file exists
//   openRead    Open file for reading
//   openWrite Open file for writing
//   close     Close (and write header) of file
//   setTrailer Set file trailer
//   getTrailer Get file trailer
//   seek        Seek to a record and offset
//
////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"

////////////////////////////////////////////////////////////////////
//
// DSPFile constructor
//
////////////////////////////////////////////////////////////////////
DSPFile::DSPFile() :
    m_type( UNKNOWN_TYPE ),
    m_elementSize( 0 ),
    m_numRecords( 0 ),
    m_recLen( 0 ),
    m_trailer( NULL ),
    m_readOnly( true ),
    m_posBeginData( 0 ),
    m_singleElement( NULL )
{
}
```

```
//////////////////////////////////////////////////////////////////////
//
// DSPFile destructor
//
//////////////////////////////////////////////////////////////////////
DSPFile::~DSPFile()
{
    close();
    empty();
}

//////////////////////////////////////////////////////////////////////
//
// empty()
//   Clears data members
//
//////////////////////////////////////////////////////////////////////
void DSPFile::empty()
{
     m_type = UNKNOWN_TYPE;
     m_elementSize = 0;
     m_numRecords = 0;
     m_recLen = 0;
     m_readOnly = true;

     if( m_singleElement != NULL )
     {
       delete [] m_singleElement;
       m_singleElement = NULL;
     }

     if( m_trailer != NULL )
     {
       delete [] m_trailer;
       m_trailer = NULL;
     }
     if( m_fs.is_open() )
       m_fs.close();
}

//////////////////////////////////////////////////////////////////////
//
// isFound( const char *name )
//   Check if file exists.
//
//  Throws:
//   DSPException
```

```
//
//  Returns:
//   true -- File exists
//   false -- File not found
//
////////////////////////////////////////////////////////////////
bool DSPFile::isFound( const char *name )
{
     // Validate filename
     if( name == NULL )
         throw DSPParamException();

     // Open the file
     fstream fs;
     fs.open( name, ios::in | ios::nocreate | ios::binary );
     if( fs.fail() )
         return false;

     fs.close();
     return true;
}
```

The first four structure elements of **DSPFile** are the same as the file header structure **HEADER** and can be written directly to or from the file. The functions **openWrite** and **openRead** (described in Section 3.2.1) both create this structure, and the other disk data format routines use it to get the information required for each operation.

Section 3.2.1 describes the formatted open routines used to open files for data read or write of a specified data type and size. Section 3.2.2 describes the read and write functions used to read or write one record at a time. Section 3.2.3 describes the functions used to read and write the optional trailer string. All of the routines described in these sections are contained in the file DISKIO.CPP on the accompanying disk. The header file DISK.H contains the structure **DSPFile** and the function prototypes for all of the disk functions. A complete example of the use of these formatted disk access routines is presented in Section 3.3.

3.2.1 Open Formatted Data File Routines

Listing 3.3 shows the **openWrite** and **close** member functions source code. This function is used to open a disk file and create the header section of the DSP data format. The **openWrite** function is part of the **DSPFile** class and requires a character string containing a valid file name to be used to open a file by a call to **fopen** (an invalid file name will throw an exception). The type of data in the **DSPFile** structure is set by the **close** member function.

```
//////////////////////////////////////////////////////////////////////
//
// openWrite( const char *name )
//   Open the file for writing.
//   The file is created, a blank header is
//   written out, and the file pointer is placed
//   on the first element where data will go.
//   Calls to write data move the file pointer.
//
//  Throws:
//   DSPException
//
//////////////////////////////////////////////////////////////////////
void DSPFile::openWrite( const char *name )
{
    // Validate filename
    if( name == NULL )
        throw DSPParamException( "No file name" );

    // Check state of file
    if( m_fs.is_open() )
        throw DSPFileException( "Already opened" );

    // Clear existing data
    empty();

    // Create a new DSPFile
    m_fs.open( name, ios::out | ios::trunc | ios::binary );
    if( m_fs.fail() )
        throw DSPFileException( "Cannot open for writing" );

    // Write blank header from DSPFile structure
    if( m_recLen > USHRT_MAX )
    {
        // Must write vector out as a LONG_VECTOR (32-bits)
        BYTE type = m_type | LONG_VECTOR;

        // Write out header
        m_fs.write( (BYTE *)&type, sizeof( type ) );
        m_fs.write( (BYTE *)&m_elementSize, sizeof( m_elementSize ) );

        // The m_numRecords field is combined with the
        // m_numRecords field as a 32-bit integer
```

LISTING 3.3 Function **openWrite** and **close** used to create a formatted data file for writing. (*Continued*)

```
        m_fs.write( (BYTE *)&m_recLen, sizeof( m_recLen ) );
    }
    else
    {
        // Write out header
        m_fs.write( (BYTE *)&m_type, sizeof( m_type ) );
        m_fs.write( (BYTE *)&m_elementSize, sizeof( m_elementSize ) );

        // Two unsigned shorts specify the number
        // of records and the record length
        unsigned short s = m_numRecords;
        m_fs.write( (BYTE *)&s, sizeof( s ) );
        s = m_recLen;
        m_fs.write( (BYTE *)&s, sizeof( s ) );
    }
    if( m_fs.fail() )
        throw DSPFileException( "Writing header" );

    // Begin data here
    m_posBeginData = m_fs.tellp();

    // File creation succeeded
    m_readOnly = false;
}

////////////////////////////////////////////////////////////////////////////
//
// close()
//   Close file. Write header and trailer if
//   file was opened with openWrite.
//
//  Throws:
//   DSPException
//
////////////////////////////////////////////////////////////////////////////
void DSPFile::close()
{
   if( m_fs.is_open() == false )
      return;

   if( m_readOnly == false && m_elementSize != 0 )
   {
       if( m_recLen == 0 )
       {
```

LISTING 3.3 (*Continued*)

```
        // File created sequentially by writeElement
        m_numRecords = 1;

        // Seek to the end of data
        m_fs.seekp( 0L, ios::end );

        // Get current position
        m_recLen = (int)(
            ( m_fs.tellp() - m_posBeginData ) /
            m_elementSize );
    }

    // Seek to the beginning of the header
    m_fs.seekp( 0L, ios::beg );

    // Update LONG_VECTOR when writing header
    if( m_recLen > USHRT_MAX )
    {
        // File format doesn't support matrices larger than 65535x65535
        if( m_numRecords != 1 )
        {
            m_fs.close();
            throw DSPFileException( "Matrix too large for file format" );
        }

        // LONG_VECTOR
        BYTE type = m_type | LONG_VECTOR;

        // Write out header
        m_fs.write( (BYTE *)&type, sizeof( type ) );
        m_fs.write( (BYTE *)&m_elementSize, sizeof( m_elementSize ) );

        // The m_numRecords field is combined with the
        // m_numRecords field as a 32-bit integer
        m_fs.write( (BYTE *)&m_recLen, sizeof( m_recLen ) );
    }
    else
    {
        // Write out header
        m_fs.write( (BYTE *)&m_type, sizeof( m_type ) );
        m_fs.write( (BYTE *)&m_elementSize, sizeof( m_elementSize ) );

        // Two unsigned shorts specify the number
        // of records and the record length
```

LISTING 3.3 (*Continued*)

```
            unsigned short s = m_numRecords;
            m_fs.write( (BYTE *)&s, sizeof( s ) );
            s = m_recLen;
            m_fs.write( (BYTE *)&s, sizeof( s ) );
        }
        if( m_fs.fail() )
        {
            m_fs.close();
            throw DSPFileException( "Writing header" );
        }

        // Write out trailer
        if( m_trailer != NULL )
        {
          streampos posTrailer = m_elementSize * m_numRecords * m_recLen;
            m_fs.seekp( posTrailer, ios::cur );
            m_fs.write( m_trailer, strlen( m_trailer ) + 1 );
            if( m_fs.fail() )
            {
                m_fs.close();
                throw DSPFileException( "Writing trailer" );
            }
        }
    }
    m_fs.close();
    empty();
}
```

LISTING 3.3 (*Continued*)

The **openRead** function (shown in Listing 3.4) is similar to the **openWrite** function except that all of the **DSPFile** structure is filled in by the **openRead** function, since the size of the data and type of data are contained in the header of the specified file. The **openRead** function operates on a **DSPFile** structure in the same way that **openWrite** does. The file is opened in binary update mode so that records or the trailer can be written as well as read using the formatted data access routines described in the next section.

3.2.2 Formatted Data Access Routines

This section describes the **read** and **write** member functions used to access the data section of the DSP data format. In each case, the user must first call **openRead** or **openWrite** with the proper parameters in order to create a valid **DSPFile** structure for use by these routines. The member function **seek** can be used with the **read** and **write**

```
////////////////////////////////////////////////////////////////
//
// openRead( const char *name )
//   Open the file for reading.
//   When the file is opened, the header and trailer
//   are read into data members and the file pointer
//   is placed on the first element of the data.
//   Calls to retrieve data move the file pointer.
//
//  Throws:
//   DSPException
//
////////////////////////////////////////////////////////////////
void DSPFile::openRead( const char *name )
{
    // Validate filename
    if( name == NULL )
        throw DSPParamException( "No file name" );

    // Check state of file
    if( m_fs.is_open() )
        throw DSPFileException( "Already opened" );

    // Clear existing data
    empty();

    // Try to open the file but do not create it
    m_fs.open( name, ios::in | ios::binary | ios::nocreate );
    if( m_fs.fail() )
        throw DSPFileException( "Cannot open for reading" );

    // File exists -- read information into DSPFile structure
    m_fs.read( (BYTE *)&m_type, sizeof( m_type ) );
    m_fs.read( (BYTE *)&m_elementSize, sizeof( m_elementSize ) );

    // Check for long vector type
    if( m_type & LONG_VECTOR )
    {
        // The m_numRecords field is combined with the
        // m_numRecords field as a 32-bit integer
        m_numRecords = 1;
        m_fs.read( (BYTE *)&m_recLen, sizeof( m_recLen ) );
        // Mask out LONG_VECTOR bit
        m_type &= ~LONG_VECTOR;
```

LISTING 3.4 Function **openRead** used to open a formatted data file for reading. (*Continued*)

```
    }
    else
    {
        // Two unsigned shorts specify the number
        // of records and the record length
        unsigned short s;
        m_fs.read( (BYTE *)&s, sizeof( s ) );
        m_numRecords = s;
        m_fs.read( (BYTE *)&s, sizeof( s ) );
        m_recLen = s;
    }
    if( m_fs.fail() )
        throw DSPFileException( "Reading header" );

    // Skip over binary data to get trailer size
    m_posBeginData = m_fs.tellg();
    m_fs.seekg( 0L, ios::end );
    streampos posEndTrailer = m_fs.tellg();

    // Subtract data size to get trailer length
    streamoff dataSize = m_elementSize * m_numRecords * m_recLen;

    // Validate header
    if( dataSize > posEndTrailer - m_posBeginData )
        throw DSPFileException( "Invalid header" );

    streamoff trailerSize = posEndTrailer - m_posBeginData - dataSize;

    // Allocate trailer
    if( trailerSize != 0L )
    {
        // Seek to beginning of trailer from end of file
        m_fs.seekg( -trailerSize, ios::end );

        // Allocate trailer
        m_trailer = new char[ trailerSize + 1 ];
        if( m_trailer == NULL )
            throw DSPMemoryException();

        // Read trailer
        m_fs.read( m_trailer, trailerSize );
        if( m_fs.fail() )
            throw DSPFileException( "Reading trailer" );

        // Make sure that string terminates
```

LISTING 3.4 (*Continued*)

```
        m_trailer[trailerSize] = 0;
    }

    // Allocate buffer for readElement
    m_singleElement = new BYTE[ m_elementSize ];
    if( m_singleElement == NULL )
        throw DSPMemoryException();

    // Place pointer at beginning of data
    seek();
    m_readOnly = true;
}
```

LISTING 3.4 (*Continued*)

functions to allow true random access of the records stored in the DSP data format. Although random access of DSP data is rarely required, the **seek** function is provided to make the data access functions general purpose.

Listing 3.5 shows the source code for the **read** member function, and Listing 3.6 shows the source code for the **write** member function. The function consists of a single call to the standard C++ function **fread** with the parameters for **fread** coming from the **DSPFile** structure. No value is returned by the function, so it is declared **void.** The

```
////////////////////////////////////////////////////////////
//
// readElement( Type& element )
//   Reads single element from file.
//
//  Note:
//   Can only be used on files with one record.
//
//  Throws:
//   DSPException
//
////////////////////////////////////////////////////////////
template <class Type>
void readElement( Type& element )
{
    // Validate state of file
    if( m_fs.is_open() == false)
        throw DSPFileException( "File not opened" );
```

LISTING 3.5 Member function **readElement** and **read** used to read data from a DSP data file of the type specified by the **DSPFile** structure. (*Continued*)

```
    if( m_readOnly == false )
        throw DSPFileException( "Not opened for reading" );

    if( m_recLen == 0 || m_numRecords == 0 )
        throw DSPFileException( "No data in file" );

    if( m_numRecords != 1 )
        throw DSPFileException( "File has more than one record" );

    if( m_type == UNKNOWN_TYPE )
        throw DSPFileException( "Cannot read from unknown type" );

    // Read single element
    m_fs.read( m_singleElement, m_elementSize );
    if( m_fs.fail() )
        throw DSPFileException( "Failure reading next element" );

    // Convert it to type
    convBuffer( &element, m_singleElement, m_type, 1 );
}
////////////////////////////////////////////////////////////////
//
// read( Vector<Type>& vec )
//   Reads vector from file. Converts from C DSP_FILE type
//   to type of vector.
//
//  Throws:
//   DSPException
//
////////////////////////////////////////////////////////////////
template <class Type>
void read( Vector<Type>& vec )
{
    // Validate state of file
    if( m_fs.is_open() == false )
        throw DSPFileException( "Not opened" );

    if( m_readOnly == false )
        throw DSPFileException( "Not opened for reading" );

    // Get current file pointer position
    int recoff = ( m_fs.tellg() - m_posBeginData ) / m_elementSize;
    if( recoff >= m_recLen * m_numRecords )
        throw DSPFileException( "Read past end of data" );
```

LISTING 3.5 (*Continued*)

```
    // Calculate elements to read until next record starts
    recoff = m_recLen - ( recoff % m_recLen );

    // Allocate vector for data
    vec.setLength( recoff );

    if( m_type == convType( typeid( Type ) ) )
    {

        // Read data directly into vector buffer without conversion
       m_fs.read( (BYTE *)vec.m_data, sizeof( Type ) * recoff );
        if( m_fs.fail() )
            throw DSPFileException( "Reading vector" );
    }
    else
    {
        // Read data into temporary buffer then convert
        BYTE *rowData = new BYTE[ m_elementSize * recoff ];
        if( rowData == NULL )
            throw DSPMemoryException();

        // Read row in
        m_fs.read( rowData, m_elementSize * recoff );
        if( m_fs.fail() )
        {
            delete [] rowData;
            throw DSPFileException( "Reading vector" );
        }
        // Convert it to vector type
        try
        {
            convBuffer( vec.m_data, rowData, m_type, recoff );
        }
        catch( DSPException& e )
        {
            delete [] rowData;
            throw e;
        }
        delete [] rowData;
    }
}
```

LISTING 3.5 (*Continued*)

```
////////////////////////////////////////////////////////////////
//
// writeElement( const Type& element )
//   Writes single element to file.
//
//  Note:
//   Can only be used on files with no data or single records.
//
//  Throws:
//   DSPException
//
////////////////////////////////////////////////////////////////
template <class Type>
void writeElement( const Type& element )
{
    // Validate state of file
    if( m_fs.is_open() == false )
        throw DSPFileException( "Not opened" );

    if( m_readOnly )
        throw DSPFileException( "Not opened for writing" );

    DSPFILETYPE dft = convType( typeid( Type ) );
    if( dft == UNKNOWN_TYPE )
        throw DSPFileException( "Writing unknown type" );
    else if( m_type == UNKNOWN_TYPE )
    {
        // Set type to data coming in
        m_type = dft;
        m_elementSize = sizeof( Type );
    }
    else if( m_type != dft )
        throw DSPFileException( "Can only write one type" );

    if( m_numRecords > 1 )
        throw DSPFileException( "Can only write to single record" );

    // We can only write vectors when using writeElement
    m_numRecords = 1;

    // Write single element
    m_fs.write( (BYTE *)&element, m_elementSize );
```

LISTING 3.6 Member function **writeElement** and **write** used to output data to a DSP data file. (*Continued*)

```
    if( m_fs.fail() )
        throw DSPFileException( "Writing single data element" );
}
////////////////////////////////////////////////////////////////
//
// write( const Vector<Type>& vec )
//   Writes vector to file. File will be the same type
//   as the vector.
//
//  Throws:
//   DSPException
//
////////////////////////////////////////////////////////////////
template <class Type>
void write( const Vector<Type>& vec )
{

    // Validate state of file
    if( m_fs.is_open() == false )
           throw DSPFileException( "opened" );

    if( m_readOnly )
           throw DSPFileException( "Not opened for writing" );

    // Validate input
    if( vec.isEmpty() )
            throw DSPFileException( "Writing empty vector" );

    DSPFILETYPE type = convType( typeid( Type ) );
    if( m_numRecords == 0 )
    {
           // First data written to file
           m_type = type;
           m_elementSize = sizeof( Type );
           m_numRecords = 0;
           m_recLen = vec.length();
    }
    else
    {
          // Data already written to file ( check type and length )
          if( m_type != type )
              throw DSPFileException( "Vector of different type than file" );
          if( vec.length() != m_recLen )
              throw DSPFileException( "Vector of different length" );
    }
```

LISTING 3.6 (*Continued*)

```
    // Write out data
    m_fs.write( (BYTE *)vec.m_data, sizeof( Type ) * m_recLen );
    if( m_fs.fail() )
        throw DSPFileException( "Writing vector" );
    m_numRecords++;
}
```

LISTING 3.6 (*Continued*)

pointer passed as the parameter of the **read** function points to the data area where the vector data is to be stored by the **read** function. The data area is allocated by the calling program. If the type of vector allocated by the calling program is different from the **DSPFile** type, the **read** function converts the disk data to the type of the vector passed in. If an error occurs in reading the file data, the function shows an exception with an error message that can be displayed by the calling program before execution stops. The member function **write** (see Listing 3.5) is almost identical to **read** except that a call to **fwrite** is used and the data pointer is used as the source of data instead of the destination.

The program shown in Listing 3.7 (file WRRECS.CPP) is an example of the use of the **write** function and the DSP file functions. The 1000-element signed integer array is written to the file OUTPUT.DAT along with the header. In this case, the integer array **dataOut** is defined in the program and set to an inverted triangular waveform by the statement **dataOut[i] = abs(i-500);** in the **for** loop.

The DSP data file OUTPUT.DAT can then be read using the program RDRECS.CPP shown in Listing 3.8. The error checking on the record length and data

```
//////////////////////////////////////////////////////////////
//
// wrrecs.cpp - Write records to a DSPFile
//   Program to demonstrate writing single records
//   to file "output.dat".
//
//////////////////////////////////////////////////////////////
#include <math.h>
#include "dsp.h"
#include "disk.h"
//////////////////////////////////////////////////////////////
//
// int main()
//
```

LISTING 3.7 Program WRRECS.CPP, which illustrates the use of **dspfile.write** to write two records to a DSP data file called OUTPUT.DAT. (*Continued*)

```
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        // File to open
        DSPFile dspfile;

        // Open file "output.dat" for writing
        dspfile.openWrite( "output.dat" );

        // Vector to write to file
        Vector<short int> dataOut( 1000 );

        // Fill vector with data
        for( int i = 0; i < dataOut.length(); i++ )
            dataOut[i] = abs( i - 500 );

        // Write first record to file
        dspfile.write( dataOut );

        // Change data in vector
        for( i = 0; i < dataOut.length(); i++ )
            dataOut[i] = i;

        // Write second record to file
        dspfile.write( dataOut );

        // Close file
        dspfile.close();

        cout << "File \"output.dat\" contains two records\n";
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 3.7 (*Continued*)

```
////////////////////////////////////////////////////////////////
//
// rdrecs.cpp - Read two records from a DSPFile
//   Program to test wrrecs.cpp by reading
//   two single records out of file "output.dat".
//
////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
////////////////////////////////////////////////////////////////
//
// int main()
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        // File to open
        DSPFile dspfile;

        // Declare Vector variables to load
        Vector<short int> r1;
        Vector<short int> r2;

        // Open file "output.dat" for reading
        dspfile.openRead( “output.dat” );
        if( dspfile.getType() != SIGNED_SHORT )
            throw DSPFileException( "Wrong data type in file" );

        // Read first and second records of file
        dspfile.read( r1 );
        dspfile.read( r2 );

        // Verify length
        if( r1.length() != 1000 || r2.length() != 1000 )
          throw DSPFileException( "Wrong record length in data file" );

        // Close file
        dspfile.close();
```

LISTING 3.8 Program RDRECS.CPP which illustrates the use of **dspfile.read** to read two records from a DSP data file called OUTPUT.DAT. (*Continued*)

```
        cout << "Read two records from file \"output.dat\"\n";
    }
    catch( DSPException& e )
    {

        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 3.8 (*Continued*)

type is not required but is important to ensure that no more than 1000 integers are read into the **r1** and **r2** arrays. The data file OUTPUT.DAT could have any type of data of any length stored in its data area (if it was not written by the **write** example shown above).

The **read** and **write** member functions are designed to allow general-purpose formatted reads and writes of any type of DSP disk data. In many cases, it is desirable to process the data using one type. For example, most FFT routines (covered in detail in Chapter 5) require floating-point arrays of data. Thus, if a program to do FFTs on a DSP data file was written using **read,** only DSP files with float data (type = 3) could be processed without conversion. The use of such a program is, therefore, quite limited, since data from analog to digital converters is almost always in an integer format. The program RDFREC.CPP (shown in Listing 3.9) solves this problem by using the convert feature in the read member function, which will convert whatever type of data is stored in the DSP data file to a **float** array. The **float** array is allocated by

```
Vector<float> floatVec;
```

```
//////////////////////////////////////////////////////////////
//
// rdfrec.cpp - Convert record to floating point
//   Program to read first record from a DSPFile
//   and convert it to a float vector.
//
//////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
```

LISTING 3.9 Program RDFREC.CPP, which illustrates the use of **dspfile.read** to read and display **float** data. (*Continued*)

```
//////////////////////////////////////////////////////////////////
//
// int main()
//
//  Returns:
//   0 -- Success
//
//////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        // File to open
        DSPFile dspfile;
        String strName;

        // Get parameters from user
        do getInput( "Enter file name", strName );
        while( strName.isEmpty() );

        // Declare float vector to read
        Vector<float> floatVec;
        // Open file for reading
        dspfile.openRead( strName );

        // Note that this file doesn't contain floats
        if( dspfile.getType() != FLOAT )
        {
            cout
                << "File \""
                << strName
                << "\" doesn't contain float data.\n";
        }
        // Read first record of file as a float vector
        // The read() member function converts the data to float
        dspfile.read( floatVec );

        // Close file
        dspfile.close();

        // Display data in a column
        cout.flags( ios::showpoint );
        for( int i = 0; i < floatVec.length(); i++ )
            cout << i << " " << floatVec[i] << endl;
```

LISTING 3.9 (*Continued*)

```
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 3.9 (*Continued*)

A **float** array was selected as the returned type because the storage space requirements are not as severe as **double** arrays, and **float** values can represent all of the data types with little loss in precision. The conversion is performed by the **inline** function **convBuffer** shown in Listing 3.10 (contained in DISK.H). In cases where the size of the data becomes excessive or precision is lost, the data type of the incoming file should be checked first and if the wrong type is stored in the file, an exception can be generated.

```
////////////////////////////////////////////////////////////////
//
// convBuffer( Type *pto, void *pfrom, DSPFILETYPE dft, int len )
//   Converts buffer from C DSP_FILE type to generic Type.
//
//   Uses inline helper function CONV to copy
//   and cast from on type to another.
//
//  Throws:
//   DSPException
//
////////////////////////////////////////////////////////////////
template <class TypeTo, class TypeFrom>
inline void CONV( TypeTo *pto, TypeFrom *pfrom, int len )
{
    while( len— )
        *pto++ = (TypeTo)*pfrom++;
}
```

LISTING 3.10 Function **convBuffer** to convert a DSP data file record and convert it to a C++ vector class. (*Continued*)

```
template <class Type>
void convBuffer( Type *pto, void *pfrom, DSPFILETYPE dft, int len )
{
    if( pto == NULL || pfrom == NULL || len <= 0 )
        throw DSPParamException( "convBuffer received no data" );
    switch( dft )
    {
    // C DSP_FILE types
   case UNSIGNED_CHAR: CONV( pto, (unsigned char *)pfrom, len ); break;
   case UNSIGNED_SHORT: CONV( pto, (unsigned short *)pfrom, len ); break;
   case UNSIGNED_LONG: CONV( pto, (unsigned long *)pfrom, len ); break;
   case FLOAT:        CONV( pto, (float *)pfrom, len ); break;
   case SIGNED_CHAR:   CONV( pto, (signed char *)pfrom, len ); break;
   case SIGNED_SHORT: CONV( pto, (signed short *)pfrom, len ); break;
   case SIGNED_LONG:   CONV( pto, (signed long *)pfrom, len ); break;
   case DOUBLE:       CONV( pto, (double *)pfrom, len ); break;

    // Additional types for C++
    case UNSIGNED_INT: CONV( pto, (unsigned int *)pfrom, len ); break;
    case SIGNED_INT:    CONV( pto, (signed int *)pfrom, len ); break;
    case COMPLEX:       CONV( pto, (Complex *)pfrom, len ); break;

    default:
        throw DSPFileException( "convBuffer of unknown type" );
    }
}
```

LISTING 3.10 (*Continued*)

3.2.3 Trailer Access Routines

Listing 3.11 shows the member functions **getTrailer** and **setTrailer.** These functions are used to get or set the trailer of an existing formatted disk file that has already been opened using **read.** A reference to the beginning of a character array containing the complete trailer is returned by **getTrailer.** For example, to display the trailer of a DSP file the program RDTRAIL.CPP shown in Listing 3.12 could be used.

This simple program reads the filename from the user with the **getInput** function, opens the file using the **read** function, and then displays the trailer string using the **getTrailer** function. The **setTrailer** member function is similar to the **getTrailer** function except that it writes to the **DSPFile** structure. For example, the program WRTRAIL.CPP shown in Listing 3.13 writes the two 1,000-element integer records to the file OUTPUT.DAT (as before) and then adds a trailer string using **setTrailer.**

```
////////////////////////////////////////////////////////////////
//
// getTrailer( String& str )
//   Get the DSPFile trailer into str.
//
//  Throws:
//   DSPException
//
////////////////////////////////////////////////////////////////
void DSPFile::getTrailer( String& str )
{
    str.empty();
    if( m_fs.is_open() == false )
        throw DSPFileException( "Not opened" );

    if( m_readOnly == false )
        throw DSPFileException( "Not opened for reading" );

    str = m_trailer;
}

////////////////////////////////////////////////////////////////
//
// setTrailer( const char *trailer )
//   Set the DSPFile trailer to trailer.
//
//  Throws:
//   DSPException
//
////////////////////////////////////////////////////////////////
void DSPFile::setTrailer( const char *trailer )
{
    if( m_fs.is_open() == false )
        throw DSPFileException( "Not opened" );

    if( m_readOnly )
        throw DSPFileException( "Not opened for writing" );

    if( m_trailer != NULL )
    {
        delete [] m_trailer;
        m_trailer = NULL;
    }
```

LISTING 3.11 Functions **getTrailer** and **setTrailer** used to read and write the trailer of a formatted DSP data file. (*Continued*)

```
    if( trailer != NULL )
    {
        // Allocate new trailer
        m_trailer = new char[ strlen( trailer ) + 1 ];
        if( m_trailer == NULL )
            throw DSPMemoryException();

        // Copy trailer
        strcpy( m_trailer, trailer );
    }
}
```

LISTING 3.11 (*Continued*)

```
//////////////////////////////////////////////////////////////////
//
// rdtrail.cpp - Read trailer from a DSPFile
//   Program to read the trailer from an open DSPFile.
//
//////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"

//////////////////////////////////////////////////////////////////
//
// int main()
//
//  Returns:
//   0 -- Success
//
//////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        // File to open
        DSPFile dspfile;
        String strName;
        String strTrailer;

        // Get parameters from user
        do getInput( "Enter file name", strName );
```

LISTING 3.12 Program RDTRAIL.CPP to read and display a DSP data file trailer. (*Continued*)

```
        while( strName.isEmpty() );

        // Open file for reading
        dspfile.openRead( strName );

        // Read trailer into string
        dspfile.getTrailer( strTrailer );

        // Close file
        dspfile.close();

        // Write trailer out to screen
        if( strTrailer.isEmpty() )
            cout << "No trailer in file \"" << strName << "\"\n";
        else
            cout << "Trailer:\n" << strTrailer << endl;
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 3.12 (*Continued*)

```
////////////////////////////////////////////////////////////////
//
// wrtrail.cpp - Write records and trialer to a DSPFile
//   Program to demonstrate writing single records
//   to file "output.dat" along with a trailer.
//
////////////////////////////////////////////////////////////////
#include <math.h>
#include "dsp.h"
#include "disk.h"
```

LISTING 3.13 Program WRTRAIL.CPP to write a DSP data file trailer. (*Continued*)

```
////////////////////////////////////////////////////////////////
//
// int main()
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        // File to open
        DSPFile dspfile;
        String strTrailer;

        // Open file "output.dat" for writing
        dspfile.openWrite( "output.dat" );

        // Vector to write to file
        Vector<short int> dataOut( 1000 );

        // Fill vector with data
        for( int i = 0; i < dataOut.length(); i++ )
            dataOut[i] = abs( i - 500 );

        // Write first record to file
        dspfile.write( dataOut );

        // Change data in vector
        for( i = 0; i < dataOut.length(); i++ )
            dataOut[i] = i;

        // Write second record to file
        dspfile.write( dataOut );

        // Create trailer
        strTrailer = "File \"output.dat\" contains two records:\n";
        strTrailer += "First record is an inverted triangle\n";
        strTrailer += "Second record is a ramp\n";
        cout << strTrailer;

        // Set files trailer
        dspfile.setTrailer( strTrailer );
```

LISTING 3.13 (*Continued*)

```
        // Close file
        dspfile.close();
    }
    catch( DSPException& e )
    {

        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 3.13 (*Continued*)

3.3 GRAPHIC DISPLAY OF DATA

This section describes a very useful program for displaying the disk data format graphically and also gives an example of the use of disk data format routines. The graphic display program WINPLOT (see Appendix B) on the accompanying floppy disk is made to work with most IBM-PCs running Windows or any device that is compatible with this computer type. The highest-resolution monochrome mode was selected to give best display on the largest number of monitors. If this program is not compatible with the graphics device you wish to use, a plotting program for your graphics device using the **read** member functions will be required.

Listing 3.14 shows the program MKWAVE, which is an example program using the formatted disk write functions (described in Section 3.2) and user interface functions (described in Section 3.1). This program creates a discrete time waveform that is the sum of a number of cosine waves. Each cosine wave has the same amplitude and a different frequency. The first part of the program declares the variables to be used in the main program and inputs from the user the number of samples to generate and the number of frequencies in the final waveform (using the function **getInput**). The next part of the program reads the frequency values from the user using the following code:

```
Vector<float> vecFreqs( freqs );
for( int i = 0; i < freqs; i++ )
{
    char format[80] = { 0 };
    sprintf( format, "Frequency %d", i );
    getInput( format, vecFreqs[i], 0.0f, 0.5f );
}
```

The string **format** (an array of 80 characters) is created by the call to the **sprintf** function. The array holds the string **"Frequency"** followed by the frequency input number

```
//////////////////////////////////////////////////////////////
//
// mkwave.cpp -- Generate waveform
//   Example use of user input and formatted DSPFile routines.
//   Generates a sampled signal which is the sum of cosine waves.
//
//  Inputs:
//   Length, number of frequencies, frequency values, and file name.
//
//  Outputs:
//   DSPFile with waveform.
//
//////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
//////////////////////////////////////////////////////////////
//
// int main()
//
// Returns:
// 0 -- Success
//
//////////////////////////////////////////////////////////////
int main()
{
    try
    {
        int samples = 0;
        getInput( "Enter number of samples", samples, 2, 10000 );

        Vector<float> vecWave( samples );
        vecWave = 0.0f;

        int freqs = 0;
        getInput( "Enter number of frequencies", freqs, 1, 20 );

        Vector<float> vecFreqs( freqs );
        for( int i = 0; i < freqs; i++ )
        {
            char format[80] = { 0 };
            sprintf( format, "Frequency %d", i );
            getInput( format, vecFreqs[i], 0.0f, 0.5f );
        }
```

LISTING 3.14 Program MKWAVE used to create a set of samples equal to the sum of a number of cosine waves. (*Continued*)

```
        makeWave( vecWave, vecFreqs );

        // Get filename
        String strName;
        do getInput( "Enter file to create", strName );
        while( strName.isEmpty() );

        // Open file for writing
        DSPFile dspfile;
        dspfile.openWrite( strName );

        // Store data in DSPFile
        dspfile.write( vecWave );

        // Write trailer
        static char buf[300];
        sprintf(
            buf,
            "\nSignal of length %d equal to the sum of %d\n"
            "cosine waves at the following frequencies:\n",
            vecWave.length(),
            vecFreqs.length() );
        String str = buf;
        for( i = 0; i < vecFreqs.length(); i++ )
        {
            sprintf( buf, "f/fs = %f\n", vecFreqs[i] );
            str += buf;
        }

        // Write trailer to display
        cout << str;

        // Add trailer to file
        dspfile.setTrailer( str );

        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 3.14 (*Continued*)

being requested (specified by the **for** loop index, i). The **getInput** function is then used to get the values of each frequency from the user. The limits of each frequency input are from 0 to 0.5 because the frequencies are specified in terms of a normalized sampling rate of unity. Thus, the frequency input value is the ratio of the signal frequency to the sampling rate and is always less than the aliasing limit of 0.5. Each one of the frequency values is stored in the **vecFreqs** floating-point array. The result of the user inputs (**length**, **nfreqs**, and the **freqs** array) are passed to the function **wave,** which generates the sum of cosines. The MKWAVE program then opens the formatted disk file and stores the one record of floating-point data. A trailer is written to the file describing the number of frequencies contained in the signal stored in the disk file (the trailer is also echoed to the user).

Listing 3.15 shows the function **wave**, which is contained in the same MKWAVE.CPP file as the main program shown in Listing 3.14. The main program of MKWAVE calls the wave function to generate the digital signal samples. The **wave** function performs the following operations:

```
//////////////////////////////////////////////////////////////
//
// void makeWave( Vector<float>& vecWave, const Vector<float>& vecFreq )
//   Fill vector vecWave with sum of frequencies in vecFreq.
//
//////////////////////////////////////////////////////////////
void makeWave( Vector<float>& vecWave, const Vector<float>& vecFreq )
{
    const double TWO_PI = 8.0 * atan(1.0);

    // For each frequency
    for( int i = 0; i < vecFreq.length(); i++ )
    {
        // Add the sinusoid to the waveform
        double arg = TWO_PI * vecFreq[i];
        for( int j = 0; j < vecWave.length(); j++ )
            vecWave[j] += (float)cos( j * arg );
    }

    // Normalize amplitude by the number of frequencies
    for( i = 0; i < vecWave.length(); i++ )
        vecWave[i] /= vecFreq.length();
}
```

LISTING 3.15 Function wave used to create a set of samples equal to the sum of a number of cosine waves (in MKWAVE.CPP).

1. Allocates the space required for the floating-point array that will be returned (as a pointer) to the calling function.
2. Creates each sample of the digital signal by accumulating the results of a series of cosine calculations, each with the frequency specified by the **freqs** array.
3. Normalizes the signal amplitude by dividing all of the signal samples by the number of frequencies in the sum.
4. Returns a pointer to the beginning of the signal array.

A typical dialogue with the MKWAVE program is as follows:

```
Enter number of samples to generate [2..10000] : 150
Enter number of frequencies in sum [1..20] : 2
Enter frequency #0 [0.0..0.5] : 0.02
Enter frequency #1 [0.0..0.5] : 0.4
Enter file name: WAVE.DAT
Signal of length 150 equal to the sum of 2
cosine waves at the following frequencies:
f/fs = 0.020000
f/fs = 0.400000
```

This use of the MKWAVE program creates a data file WAVE.DAT (in the format described in Section 3.2) with one 150-sample record that consists of the sum of two cosine waves at frequencies of 0.02 and 0.4. The WINPLOT graphics program can then be used to generate a plot of this DSP data file as follows:

```
WINPLOT WAVE.DAT
```

The WINPLOT program also contains a graphical user interface that allows multiple graphs to be displayed on the screen and will display each of the records as a plot of the sample value versus the sample number (starting at 0 and ending at $N-1$, where N is the number of samples in the record). For the example above, the single record in WAVE.DAT generates the graphical output shown in Figure 3.2. The 0.02 frequency appears as three cycles on the 150-sample signal, and the 0.4 frequency appears as a large amount of noise added to the lower 0.02 frequency. The high frequency appears as noise because it is quite close to the aliasing limit at 0.5 and because the WINPLOT program draws (or tries to draw) straight lines between each sample causing the connected sample points to look noisy.

If more than one record is present in the data file passed to the WINPLOT program, then each record can be sequentially plotted by pressing the down arrow key after each plot. A hard copy can also be generated after the graph is displayed on the screen.

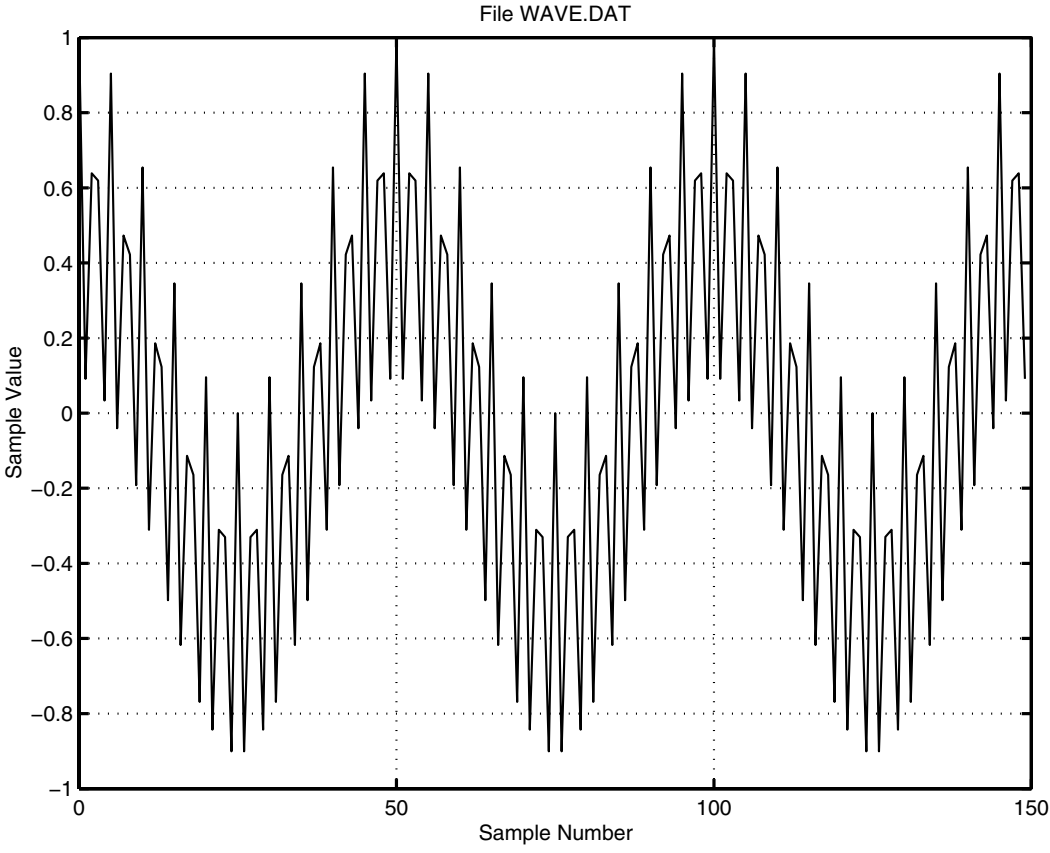


FIGURE 3.2 Result of using the plot program on the WAVE.DAT DSP data file.

Another way to generate hard copy of the graphic output of the PLOT program is to use the graphic print screen function built into the operating system.

3.4 EXERCISES

1. Modify the MKWAVE.C program to generate an FM chirp signal instead of a constant cosine wave. Hint: In the function wave(), change the cos(i*arg) statement to cos(0.01*i*i*arg). Generate 200 samples at an input frequency of 0.1. What frequencies will be generated by this signal? Hint: Frequency is the derivative of the phase of a signal.
2. Run program WRRECS.CPP (see Section 3.2.2) to generate integer data of an inverted triangle in the file OUTPUT.DAT. Write a new C++ program called CROP.CPP to read in the data file, and print the outputs from WRRECS program. Change the CROP.CPP source code to add the ability to start at a particular sample, and generate an output integer data of a re-

duced size. Run program CROP to read in the OUTPUT.DAT file, and generate a smaller file called CROPOUT.DAT. Use a start point of 495 and a length of 10, for example.

3. Modify the program CROP.CPP to input a "stride" value that will skip one or more samples in the input data. For example, a stride value of 1 would leave only the even number samples in input data. A stride value of 2 would leave only every third sample in input data This type of program can be used for decimation of a signal (see Chapter 4, Section 4.5).

3.5 REFERENCES ON USER INTERFACE

BAECKER, R. M., and BUXTON, W. A. S., *Readings in Human-Computer Interaction: A Multidisciplinary Approach,* Morgan Kaufmann, Los Altos, CA, 1987.

CARD, S. K., MORAN, T. P., and NEWELL, A., *Psychology of Human-Computer Interaction,* Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

KANTOWITZ, B. H., and SORKIN, R. D., *Human Factors: Understanding People-System Relationships,* John Wiley and Sons, New York, 1983.

MARTIN, J., *Design of Man-Computer Dialogues,* Prentice Hall, Englewood Cliffs, NJ, 1973.

MEHLMANN, M., *When People Use Computers: An Approach to Developing an Interface,* Prentice Hall, Englewood Cliffs, NJ, 1981.

ROCHKIND, M. J., *Advanced C Programming for Displays,* Prentice Hall, Englewood Cliffs, NJ, 1988.

SHNEIDERMAN, B., *Designing the User Interface,* Addison-Wesley, Reading, MA, 1987.

CHAPTER 4

Filtering Routines

Filtering is the most commonly used signal processing technique. Filters are usually used to remove or attenuate an undesired portion of a signal's spectrum while enhancing the desired portions of the signal. Often, the undesired portion of a signal is random noise with a frequency content different from the desired portion of the signal. Thus, by designing a filter to remove some of the random noise, the signal-to-noise ratio can be improved in some measurable way.

Filtering can be performed using analog circuits with continuous-time analog inputs or digital circuits with discrete time digital inputs. In systems where the input signal is digital samples (in music synthesis or digital transmission systems, for example), a digital filter can be used directly. If the input signal is from a sensor that produces an analog voltage or current, then an analog-to-digital (A/D) converter is required to create the digital samples. In either case, a digital filter can be used to alter the spectrum of the sampled signal, x_i, in order to produce an enhanced output, y_i. Digital filtering can be performed in either the time or frequency domain with general-purpose computers using previously stored digital samples or in real-time with dedicated hardware. In this chapter, we consider the time domain methods of implementing digital filters. Chapter 5 is concerned with frequency domain methods using the discrete Fourier transform. In both chapters, the programs introduced can be used with real-time hardware or with general-purpose computers.

Figure 4.1 shows a typical digital filter structure containing N memory elements used to store the input samples and N memory elements (or delay elements) used to store the output sequence. As a new sample comes in, the contents of each of the input memory elements are copied to the memory elements to the right. As each output sample is

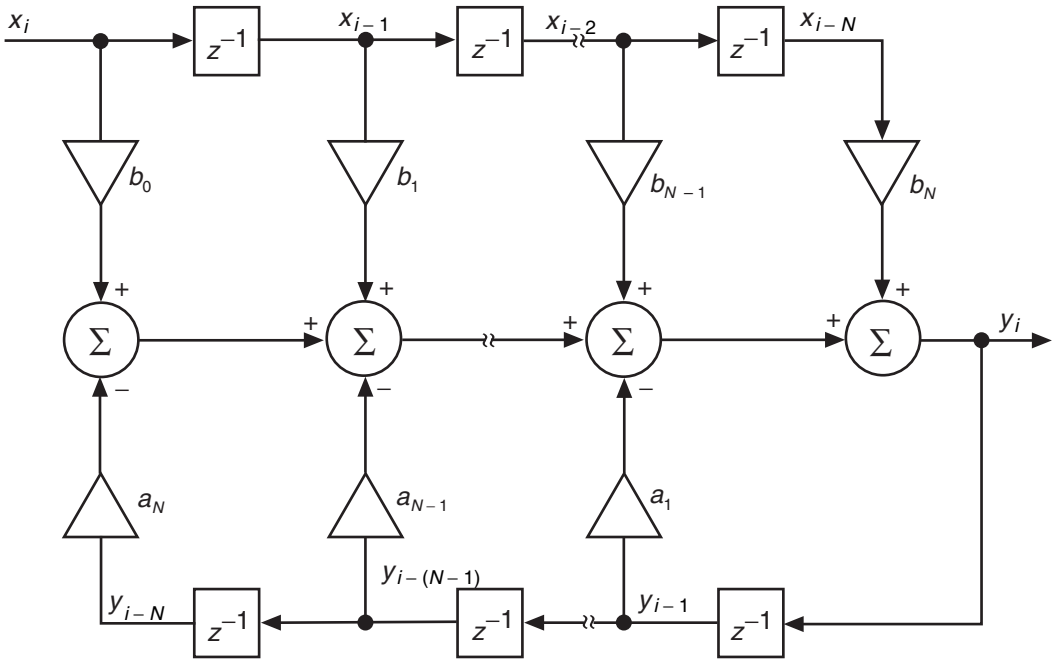


FIGURE 4.1 Filter structure of *N*th order filter. The previous *N* input and output samples stored in the delay elements are used to form the output sum.

formed by accumulating the products of the coefficients and the stored values, the output memory elements are copied to the left. The series of memory elements forms a digital delay line. The delayed values used to form the filter output are called *taps* because each output makes an intermediate connection along the delay line to provide a particular delay. This filter structure implements the following difference equation:

$$y_i = \sum_{q=0}^{N} b_q x_{i-q} - \sum_{p=1}^{N} a_p y_{i-p} \tag{4.1}$$

As discussed in Chapter 1, filters can be classified based on the duration of their impulse response. Filters where the a_n terms are zero are called *finite impulse response* (FIR) *filters* because the response of the filter to an impulse (or any other input signal) cannot change *N* samples past the last excitation. FIR filters are discussed in Section 4.2 and in Chapter 1 in Section 1.3.1. Filters where one or more of the a_n terms are nonzero are *infinite impulse response* (IIR) *filters.* Because the output of an IIR filter depends on a sum of the *N* input samples as well as a sum of the past *N* output samples, the output response is essentially dependent on all past inputs. Thus, the filter output response to any finite-length input is infinite in length giving the IIR filter infinite memory. IIR filters are

discussed in Section 4.3 and in Chapter 1 in Section 1.3.3. The remaining sections of this chapter discuss several applications of FIR and IIR linear filtering (Sections 4.4 to 4.7), as well as a brief introduction to nonlinear filtering (Section 4.8).

Many of the C++ language implementations of the digital filtering functions discussed in this chapter will use a **Filter** class to store the FIR or IIR filter coefficients and the other information associated with the filter. This class is defined as follows:

```
////////////////////////////////////////////////////////////////
//
// Class Filter (abstract base class)
//
////////////////////////////////////////////////////////////////
template <class Type>
class Filter
{
public:
    // Constructor
    Filter() :
     m_coef( NULL ),
     m_hist( NULL ),
     m_length( 0 )
    {
     }

    // Destructor
    ~Filter()
    {
     if( m_coef != NULL )
          delete [] m_coef;
     if( m_hist != NULL )
         delete [] m_hist;
     m_length = 0;
    }

    ////////////////////////////////////////////////////////////
    //
    // Virtual pure functions are implemented in derived classes
    //
    ////////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////
    //
    // filter( const Vector<Type>& vIn )
    //   Performs filtering on a Vector.
    //
    //  Note:
    //   Default implementation uses the virtual pure
```

```
//   function filterElement() which derived classes
//   must implement.
//
//  Returns:
//   Filtered Vector
//
////////////////////////////////////////////////////////////
virtual Vector<Type> filter( const Vector<Type>& vIn )
{
 int lenIn = vIn.length();
 Vector<Type> vOut( lenIn );
 vOut = (Type)0;

 for( int i = 0; i < vIn.length(); i++ )
   vOut[i] = filterElement( vIn[i] );

 return vOut;
}

////////////////////////////////////////////////////////////
//
// filterElement( const Type input )
//   Performs filtering sample by sample on Type.
//
// Returns:
//  Filtered element
//
////////////////////////////////////////////////////////////
virtual Type filterElement( const Type input ) = 0;

////////////////////////////////////////////////////////////
//
// reset()
//   Reset history of filter, but keep parameters.
//
////////////////////////////////////////////////////////////
virtual void reset() = 0;

// Allow fast read-only access to data
const Type *getCoefficients( int len ) const
{
  if( len != m_length )
    throw DSPBoundaryException( "Coefficients different length" );
  return m_coef;
}
// Return the number of coefficients
int length() const { return m_length; }
```

```
protected:
    // Pointer to coefficients of filter
    Type *m_coef;
    // Pointer to history of filter
    Type *m_hist;
    // Size of the filter
    int m_length;
};
```

This class consists of an integer giving the length of the filter (the number of coefficients for FIR filters and the number of second-order sections for IIR filters), a pointer to an optional floating-point array used to store the past history associated with the input or output of the filter, and a pointer to a floating-point array of coefficient values. Because this structure is used by all of the floating-point filter routines discussed in this chapter, the interpretation of these three elements depends on the type of filtering being performed. The **Filter** class is defined in the header file FILTER.H along with the IIR and FIR filter classes and the prototypes for the filtering functions described in this chapter.

4.1 DIGITAL VERSUS ANALOG FILTERS

Although this chapter is concerned with discrete time digital filtering methods, analog filters have a long and fruitful history and are still the method of choice in many cases. In fact, many digital filters are designed based on the theory developed for analog filters. For these reasons, it is important to recognize the strengths and weaknesses of analog filters when compared with digital filters.

Analog filters are circuits made of reactive and capacitive elements that create a filter by storing and releasing energy with different time constants. The input and output of a continuous-time analog filter is always coupled into its storage elements so that it will always have an infinite impulse response. Also, the circuit elements are never perfect, so an ideal transfer function (requiring lossless storage elements) is never obtained. The practical considerations of analog filter design such as impedance matching, transmission line effects, component aging, and tuning can make analog filters difficult to design, use, and manufacture. These practical limitations also limit the maximum filter order that can be reliably produced. This maximum filter order ultimately restricts bandwidth (or transition bandwidth) to some lower bound depending on the filter frequency and the type of circuit elements being used.

Because all of the operations in a *digital filter* are discrete in nature (i.e., each result is computed during a fixed time interval) and are deterministic, a digital filter of any order N (see Figure 4.1) can be constructed. Practical considerations such as the number of components, the speed at which the multiplies and adds must occur, and the number of bits used to represent the input and output ultimately determine only the cost and size of the digital filter. Digital filters can be inexpensively implemented using extremely small

geometry microelectronic techniques not available for analog circuits. In fact, filters that are very difficult to implement with analog filters (such as precise, high-order filters or linear phase filters) can be routinely implemented using digital filters.

Although digital filters can implement filters that are difficult or impossible to build using analog methods, digital filters usually consume more power and are more expensive because they require more active components. One important exception to this is when a large number of analog filters are required that can be efficiently implemented using a single digital signal processing integrated circuit. Also, the dynamic range of analog filters (the maximum signal-to-noise ratio that can be passed by the filter) can always be made to exceed the performance of any equivalent digital implementation. This is primarily because the analog-to-digital converter required at the input of an equivalent digital system is an analog circuit with far less dynamic range than the components in a properly designed analog filter. In fact, in many digital filtering systems, an analog filter is required before the analog-to-digital converter in order to reduce aliasing and improve the signal-to-noise ratio.

Analog filters can be constructed to operate at higher frequencies than digital filters. Using microwave circuit techniques, analog filters can be designed to work well above 1 gigahertz. Digital filters are limited by the available analog-to-digital converters and by the speed and complexity of the digital circuits used to implement them. Analog-to-digital converters depend on the settling time of analog circuits which limits the accuracy obtainable at a particular sampling rate. Thus, the settling time and noise in the analog circuits in the A/D converter both limit the input signal dynamic range by limiting the number of valid data bits. For example, 16-bit converters are available with sampling rates as high as 200 kHz, but only 8-bit converters are available at 500 MHz sampling rates.

4.2 FIR FILTERS

Finite impulse response filters have several properties that make them useful for a wide range of applications. A perfect linear phase response can easily be constructed with an FIR filter allowing a signal to be passed without phase distortion. FIR filters are inherently stable, so stability concerns do not arise in the design or implementation phase of development. Even though FIR filters typically require a large number of multiplies and adds per input sample, they can be implemented using fast convolution with FFT algorithms (see Chapter 5). Also, FIR structures are simpler and easier to implement with standard fixed-point digital circuits at very high speeds. The only possible disadvantage of FIR filters is that they require more multiplies for a given frequency response when compared to IIR filters and, therefore, often exhibit a longer processing delay for the input to reach the output.

During the past 20 years, many techniques have been developed for the design and implementation of FIR filters. *Windowing* is perhaps the simplest and oldest FIR design technique (see Chapter 5 for a description of some of the available windows) but is quite limited in practice. The *window design method* has no independent control over the pass-

band and stopband ripple. Also, filters with unconventional responses such as *multiple passband filters* cannot be designed. On the other hand, window design can be done with a pocket calculator and can come close to optimal in some cases.

This section discusses optimal FIR filter design with equiripple error with different weights in the passbands and stopbands. This class of FIR filters is widely used primarily because of the well-known *Remez exchange algorithm* developed for FIR filters by McClellan and Parks. The general McClellan-Parks program can be used to design filters with several passbands and stopbands, digital differentiators, and Hilbert transformers. The FIR coefficients obtained using the McClellan-Parks program can be used directly with the structure shown in Figure 4.1 (with the a_n terms equal to zero). The floating-point coefficients obtained can be implemented using floating-point arithmetic (see Section 4.2.1), or they can be rounded and used with integer arithmetic (see Section 4.2.2).

The McClellan-Parks program is available on the enclosed disk (see file REMEZ.C) or as part of many of the filter design packages available for personal computers. The program is also printed in several DSP texts (see Elliot or Rabiner and Gold). The McClellan-Parks program requires the following information:

1. Filter type—multiple passband/stopband, differentiator, or Hilbert transformer
2. Filter length—number of taps (N)
3. Number of bands
4. Band edges—the lower and upper edges for each passband and stopband as a fraction of the sampling frequency
5. Desired response for each band (1 for passbands, 0 for stopbands, for example)
6. Weight associated with each band

Because the stopband attenuation and passband ripple are not specified as an input to the McClellan-Parks program, it is often difficult to determine the filter length required for a particular filter specification. Guessing the filter length will eventually reach a reasonable solution but can take a long time. Further, if the filter length given is very far from the required value, the algorithm may not converge. For one stopband and one passband, the following approximation for the filter length (N) of an optimal lowpass filter has been developed by Kaiser:

$$N = \frac{-20 \log_{10} \sqrt{\delta_1 \delta_2} - 13}{14.6 \Delta f} + 1 \tag{4.2}$$

where:

$$\delta_1 = 1 - 10^{-A_{max}/40}$$
$$\delta_2 = 10^{-A_{stop}/20}$$
$$\Delta f = (f_{stop} - f_{pass})/f_s$$

A_{max} is the total passband ripple (in dB) of the passband from 0 to f_{pass}. If the maximum of the magnitude response is 0 dB, then A_{max} is the maximum attenuation throughout the passband. A_{stop} is the minimum stopband attenuation (in dB) of the stopband from f_{stop} to $f_s/2$. The approximation for N is accurate within about 10% of the actual required filter length (usually on the low side). The ratio of the passband error (δ_1) to the stopband error (δ_2) is entered by choosing appropriate weights for each band. Higher weighting of stopbands will increase the minimum attenuation; higher weighting of the passband will decrease the passband ripple.

As a simple example, consider the following filter specifications that specify a lowpass filter designed to remove the upper half of the signal spectrum:

Passband (f_{pass}):	0-0.2f_S
Passband ripple (A_{max}):	<0.5 dB
Stopband (f_{stop}):	0.25-0.5 f_S
Stopband attenuation (A_{stop}):	>40 dB

From these specifications we find that

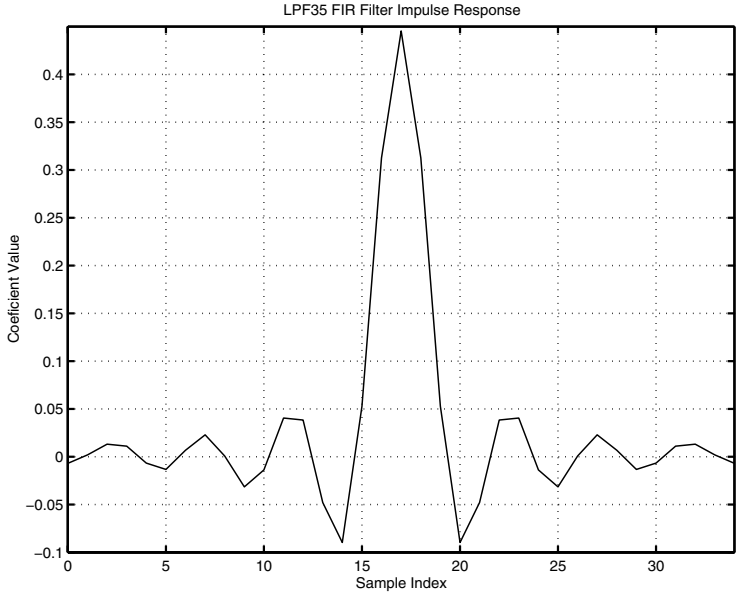
$$\delta_1 = 0.02837,$$
$$\delta_2 = 0.01,$$
$$\Delta f = 0.05.$$

The result of Equation 4.2 is then $N = 32$. After running the McClellan-Parks program with the above specifications with a stopband weighting 3 times the passband weighting (rounding the 2.837 ratio of $\delta_1 : \delta_2$ to 3 for convenience), the specifications are not quite met (A_{stop} is 38.4 dB and A_{max} = 0.61 dB). In practice, the order of the filter is not always monotonically related to the product $\delta_1\delta_2$. Sometimes a particular odd filter is much better than an even filter of greater order. After a little trial and error it was found that a 35-point filter meets the specifications with A_{stop} = 40.6 dB and A_{max} = 0.48 dB. The impulse response of this filter is shown in Figure 4.2(a). The frequency response was determined using a 1024-point fast Fourier transform (see Section 5.9 in Chapter 5) and is shown in Figure 4.2(b).

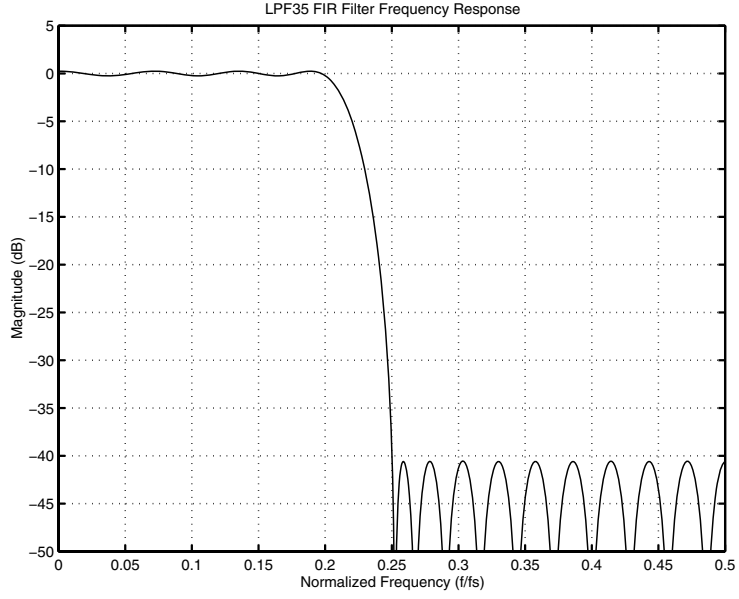
Although the passband and stopband band edges can be specified to be any value from 0 to 0.5 f_s, in practice it is often easier to use the above design equation for the filter length of a lowpass filter and then translate the lowpass filter to a different center frequency to make a bandpass or highpass filter. The new translated filter coefficients, g_n, are related to the lowpass coefficients, h_n, by the well-known frequency translation formula as follows:

$$g_n = h_n e^{j2\pi f_{shift}\, n} \tag{4.3}$$

The normalized frequency shift, f_{shift} (0 to 0.5), is the shift of the frequency response of g_n relative to the frequency response of h_n. In general, g_n is a complex function that forms a bandpass filter with a bandwidth of twice the cutoff frequency of the lowpass filter and a center frequency equal to f_{shift} (complex filters are discussed in Section 4.6). By



(a)



(b)

FIGURE 4.2 (a) Impulse response of 35-point FIR lowpass filter designed using the McClellan-Park program. (b) Frequency response of 35-point FIR lowpass filter.

taking the real part of g_n, a real bandpass filter can also be obtained when f_{shift} is greater than the lowpass filter stopband frequency, f_{stop}. A highpass filter can be obtained by setting $f_{shift} = 0.5$, which results in multiplying the h_n coefficients by a sequence of alternating ± 1. The passband of the highpass filter is from $(0.5f_s - f_{pass})$ to $0.5f_s$ and the stopband is from 0 to $(0.5f_s - f_{stop})$. The resulting impulse response and frequency response of the highpass filter obtained by translating the 35-point lowpass filter are shown in Figures 4.3(a) and 4.3(b), respectively.

4.2.1 Floating-Point FIR Filters

FIR filters can be implemented with integer or floating-point arithmetic. Floating-point implementations have the advantage that the coefficients generated by the McClellan-Parks program can be used directly with no change in the resulting frequency response. Integer implementations (see Section 4.2.2) have the advantage that the hardware required to implement integer arithmetic is usually faster and less expensive than floating-point hardware. The **FIRFilter** class shown in Listing 4.1 takes an input **Vector**, filters it, and stores the result in an output **Vector**. The filter coefficients are passed to the function using the **Filter** structure (as defined in FILTER.H). The **filter** function directly implements the following convolution equation:

$$y_i = \sum_{n=0}^{L-1} b_n x_{i-n} \tag{4.4}$$

where x_i, is the input array, y_i is the output array, and b_n is the coefficient array. Note that the order of the filter (N in Equation 4.1) is the filter length (L) minus one (it takes two coefficients to make a first-order FIR filter). The length of the filter (L) and the coefficient array must be defined in the FILTER structure, but the history pointer is not used**.** The history of the input values required by the FIR filter in this case comes from the input array itself. An implementation of FIR filtering that implements the structure shown in Figure 4.1 using a history array is discussed in Section 4.4.1, where real-time FIR filtering is described.

Normally, the direct form implementation of the convolution equation generates an output array that is $L - 1$ samples longer than the input array. The samples before and after the input samples are assumed to be zero when the input sequence overlaps only part of the filter function. The filter output exhibits a startup transient length equal to the length of the filter until the zeroes do not influence the output. A similar effect occurs when the end of the input data is approached. In digital signal processing, the startup and end responses generally do not contain useful information, so it is undesirable to generate an output array that is longer than the input array. The **FIRFilter** member function generates the same number of output samples as input samples. Figure 4.4 illustrates the FIR filter convolution used by the **FIRFilter** member function. The $L/2$ samples at the beginning of the convolution [$(L - 1)/2$ for odd L] are not calculated, so that the output samples align with the input samples. This $(L - 1)/2$ sample skip in the convolution output exactly

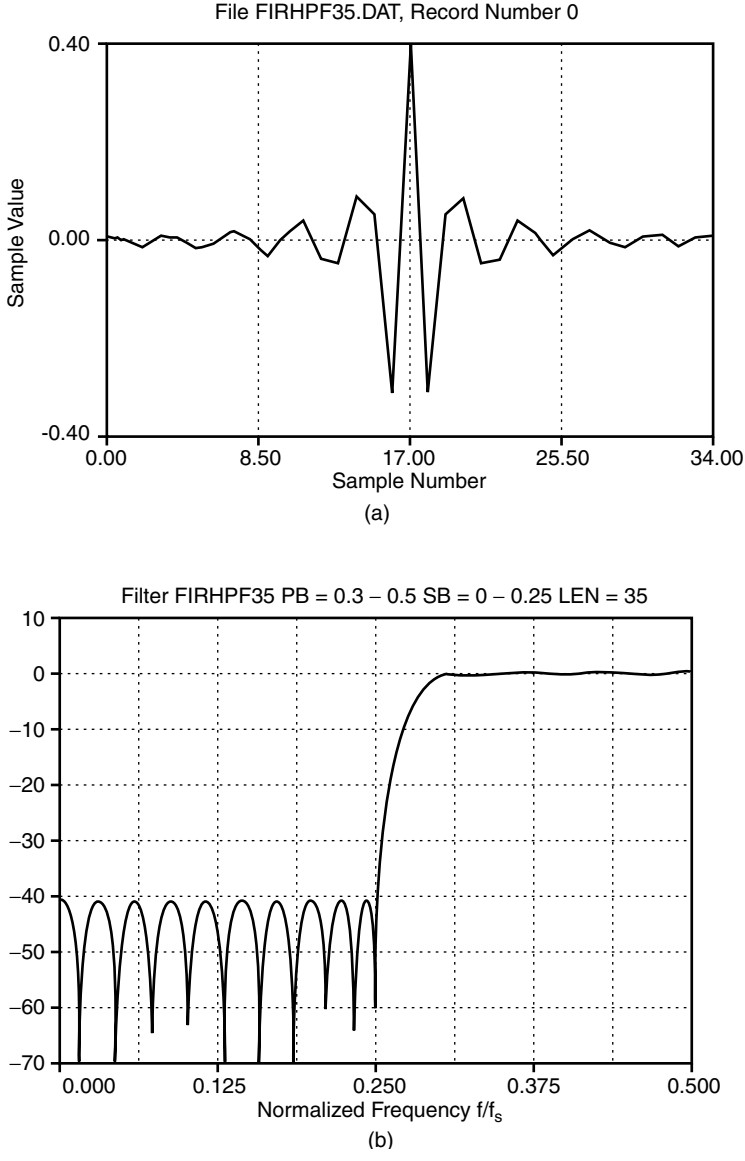


FIGURE 4.3 (a) Impulse response of 35-point FIR highpass filter obtained from the lowpass filter shown in Figure 4.2 by frequency translation. (b) Frequency response of 35-point FIR highpass filter.

```
//////////////////////////////////////////////////////////////////////
//
// Class FIRFilter -- Implements abstract base class filter
//
//////////////////////////////////////////////////////////////////////
template <class Type>
class FIRFilter : public Filter<Type>
{
public:
    // Constructor
    FIRFilter( const Type *coef, int length )
    {
     // Validate parameters
     if( length <= 0 )
         throw DSPParamException( "No filter data" );

     if( coef == NULL )
         throw DSPParamException( "Null filter coefficients" );

     // Allocate memory for filter coefficients and history
     m_coef = new Type[length];
     if( m_coef == NULL )
         throw DSPMemoryException();

     m_hist = new Type[length];
     if( m_hist == NULL )
         throw DSPMemoryException();
     m_length = length;

     // Copy coefficients
     Type *pc = m_coef;
     for( int i = 0; i < length; i++ )
         *pc++ = *coef++;
     reset();
    }

    // Implementation of abstract base class function
    virtual Vector<Type> filter( const Vector<Type>& vIn )
    {
     int lenIn = vIn.length();
     Vector<Type> vOut( lenIn );
     vOut = (Type)0;

     // Set up for coefficients
```

LISTING 4.1 Class **FIRFilter** (in FILTER.H) used to filter a vector using the **Filter** class. (*Continued*)

```
    Type *startCoef = m_coef;
    int lenCoef2 = ( m_length + 1 ) / 2;

    // Set up input data pointers
    const Type *endIn = vIn.getData( lenIn ) + lenIn - 1;
    const Type *ptrIn = vIn.getData( lenIn ) + lenCoef2 - 1;

    // Initial value of accumulation length for startup
    int lenAcc = lenCoef2;

    for(int i = 0 ; i < lenIn; i++ )
    {
        // Set up pointers for accumulation
        const Type *ptrData = ptrIn;
        const Type *ptrCoef = startCoef;

        // Do accumulation and write result
        Type acc = *ptrCoef++ * *ptrData—;
        for( int j = 1; j < lenAcc; j++ )
            acc += *ptrCoef++ * *ptrData—;
        vOut[i] = acc;

        // Check for end case
        if( ptrIn == endIn )
        {
            // One shorter each time
            lenAcc—;
            // Next coefficient each time
            startCoef++;
        }
        else
        {
            // Check for startup
            if( lenAcc < m_length )
            {
                // Add to input pointer
                lenAcc++;
            }
            ptrIn++;
        }
    }
    return vOut;
}

// Implementation of abstract base class function
```

LISTING 4.1 (*Continued*)

```
    virtual Type filterElement( const Type input )
    {

     // Start at beginning of history
     Type *ptrHist = m_hist;
     Type *ptrHist1 = m_hist;

     // point to last coefficient
     Type *ptrCoef = m_coef + m_length - 1;

     // Form output accumulation
     Type output = *ptrHist++ * *ptrCoef—;
     for( int i = 2; i < m_length; i++ )
     {
         // Update history array
         *ptrHist1++ = *ptrHist;
         output += *ptrHist++ * *ptrCoef—;
     }

     // Input tap
     output += input * *ptrCoef;
     // Last history
     *ptrHist1 = input;

     return output;
    }

    // Implementation of abstract base class function
    virtual void reset()
    {
     Type *ph = m_hist;
     for( int i = 0; i < m_length; i++ )
         *ph++ = (Type)0;
    }

    Type getHistory( int i )
    {
     if( i < 0 || i >= m_length )
         throw DSPBoundaryException( “Element outside of history” );
     return m_hist[i];
    }

private:
    // No default constructor
    FIRFilter();
};
```

LISTING 4.1 (*Continued*)

compensates for the delay of an odd-length linear phase FIR filter. Even-length FIR filters have a delay that is not an integral number of samples, so the *L*/2 sample skip is half a sample too large.

The convolution equation could be implemented directly for a particular output **y[i]** using C statements as follows:

```
y[i] = 0.0;
for(j = 0 ; j < L ; j++)
    y[i] += b[j] * x[i-j];
```

Although this will work (if i > L - 1) and is very readable, it is inefficient because of all the address arithmetic required and multiple accesses to the output array. The following code is used in **FIRFilter:**

```
Type acc = *ptrCoef++ * *ptrData—;
for( int j = 1; j < lenAcc; j++ )
    acc += *ptrCoef++ * *ptrData—;
vOut[i] = acc;
```

The first statement does the first multiply of the filter; the for loop does the remaining multiply-accumulates. If floating-point registers are available in the target processor, the variable **acc** should be declared as a register for maximum efficiency. The input data pointer (**ptrData**) is initially pointing to **vIn[(L − 1)/2]**, and the initial accumulation length (**lenAcc**) is **(L + 1)/2** (where **L** is the filter length). Thus, the first output is calculated based on the first half of the filter coefficients. As each new output sample is calculated, **lenAcc** is incremented until it equals the filter length and the startup samples are completed. A similar reduction in the accumulation length occurs when the input pointer reaches the end of the input array.

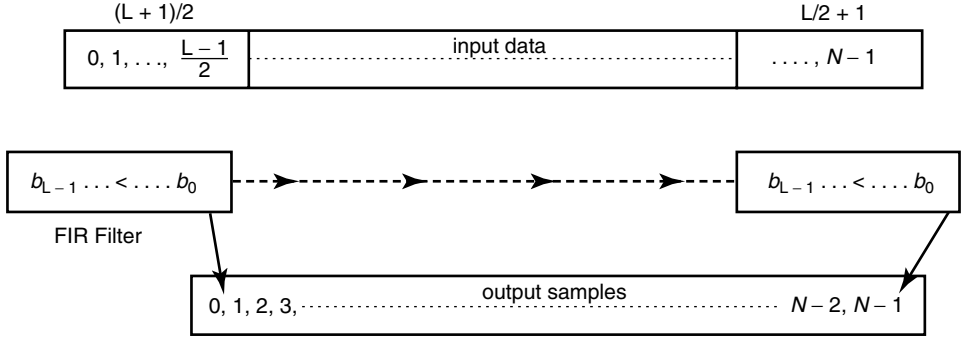


FIGURE 4.4 Illustration of the FIR filtering performed by the **FIRFilter** class. The overlap of the input data (length *N*) and the filter coefficients are shown for the first and last output samples. The number of input samples in the convolutions for the first and last output samples is indicated (integer truncation in the divide by two is assumed).

The program FIRFILT shown in Listing 4.2 uses the **FIRFilter** class to filter the records in a DSP data file. For example, the data file WAVE3.DAT shown graphically in Figure 4.5(a) can be filtered using FIRFILT as shown below:

```
Enter input file name : WAVE3.DAT

File trailer:

Signal of length 150 equal to the sum of 3
cosine waves at the following frequencies:
f/fs = 0.010000
f/fs = 0.020000
f/fs = 0.400000

Enter filter type (0 = LPF, 1 = HPF, 2 = PULSE) [0...2] : 0

Enter output file name : WAVE3FO.DAT
```

```
//////////////////////////////////////////////////////////////////////
//
// firfilt.cpp - Filter records using FIR Filters
//
//////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "filter.h"
#include "fircoefs.h"

//////////////////////////////////////////////////////////////////////
//
// int main()
//   Filters DSPFile records using a finite-impulse-response
//   filter and generate a new DSPFile containing filtered data.
//
//  Returns:
//   0 -- Success
//
//////////////////////////////////////////////////////////////////////
int main()
{
  try
  {
```

LISTING 4.2 Program FIRFILT uses the **filter** class and one of the example FIR filters to filter a DSP formatted data record. (*Continued*)

```
DSPFile dspfileIn;
DSPFile dspfileOut;
String strName;
String strTrailer;

// Open the input file
do getInput( "Enter file to filter", strName );
while( strName.isEmpty() || dspfileIn.isFound( strName ) == false );
dspfileIn.openRead( strName );
dspfileIn.getTrailer( strTrailer );

// Print trailer
if( strTrailer.isEmpty() == false )
    cout << "File trailer:\n" << strTrailer << endl;

// Get type of filter to use of several in filter.h
int typeFilter = 0;
getInput(
    "Enter type of filter ( 0 = LPF, 1 = HPF, 2 = PULSE )",
    typeFilter,
    FILTERTYPE_FIRLPF,
    FILTERTYPE_FIRPULSE );

// Create filter with coefficients chosen by user
String strType;
Filter<float> *filter = NULL;
switch( typeFilter )
{
case FILTERTYPE_FIRLPF:
    strType = "Filtered using 35 tap lowpass FIR at 0.2\n";
    filter = &FIRLPF;
    break;
case FILTERTYPE_FIRHPF:
    strType = "Filtered using 35 tap highpass FIR at 0.3\n";
    filter = &FIRHPF;
    break;
case FILTERTYPE_FIRPULSE:
    strType = "Filtered using 52 tap bandpass matched FIR \n";
    filter = &FIRPULSE;
    break;
}

// Write filtered vector out to file
do getInput( "Enter filtered output file name", strName );
while( strName.isEmpty() );
```

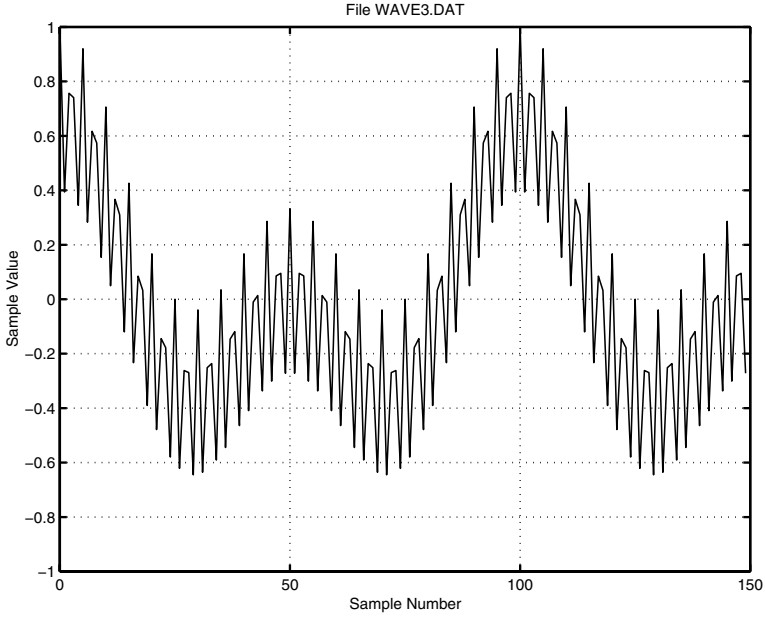
LISTING 4.2 (*Continued*)

```
    dspfileOut.openWrite( strName );

    // Filter data one record at a time
    Vector<float> vIn;
    for( int i = 0; i < dspfileIn.getNumRecords(); i++ )
    {
        // Get record
        dspfileIn.read( vIn );
        // Write filtered record
        dspfileOut.write( filter->filter( vIn ) );
    }

    // Append to trailer
    strTrailer += strType;
    dspfileOut.setTrailer( strTrailer );

    // Close files
    dspfileIn.close();
    dspfileOut.close();
  }
  catch( DSPException& e )
  {
   // Display exception
   cerr << e;
   return 1;
  }
  return 0;
}
```

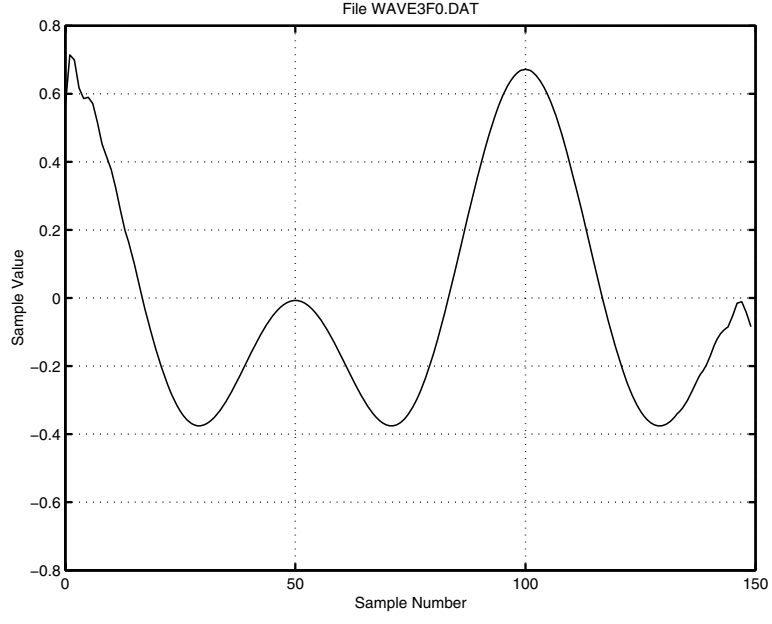
LISTING 4.2 (*Continued*)

In this case, the 35-point lowpass filter (type 0) was selected. The three filters that can be selected (lowpass, highpass, or pulse) are defined in FILTER.CPP. The pulse filter is a matched bandpass filter and will be discussed in Section 4.7.3. The result of the above lowpass filtering of the WAVE3.DAT record is shown in Figure 4.5(b). Note that the highest frequency of the WAVE3.DAT record is removed but the 0.01 and 0.02 frequencies are preserved.

Figure 4.6(a) shows a 6000-sample voice data set stored in the file CHKL.DAT. It consists of 0.75 seconds (the sampling rate of the 8-bit A/D converter was 8 kHz) of the author speaking the words "chicken little." This speech waveform contains a great deal of amplitude and frequency variation typical of speech data. Each syllable of the two words are visible (chi-ken-lit-tle) in the envelope of the signal. The 5-msec gap starting at about sample 1200 is the stop between "chi" and "ken." Figure 4.6(b) shows the result of filtering the CHKL.DAT data file with the 35-point highpass FIR filter (selection 1) using the

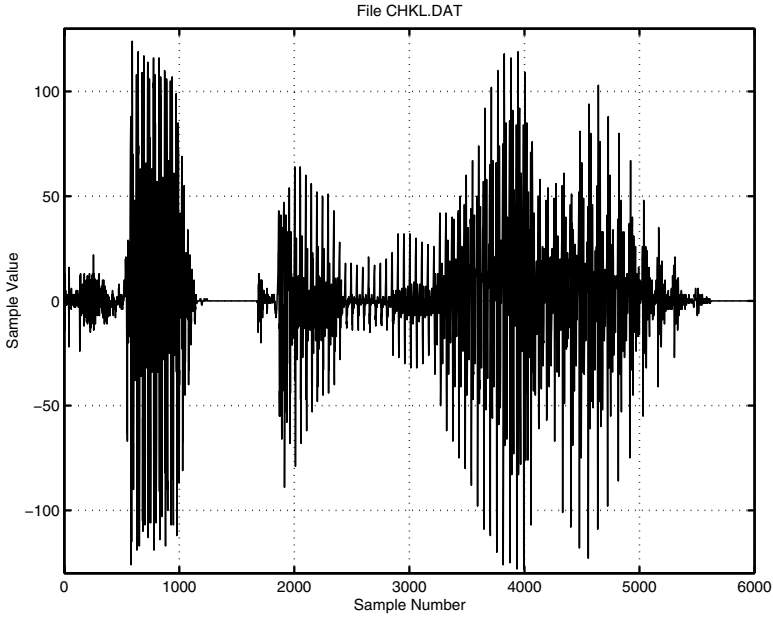


(a)

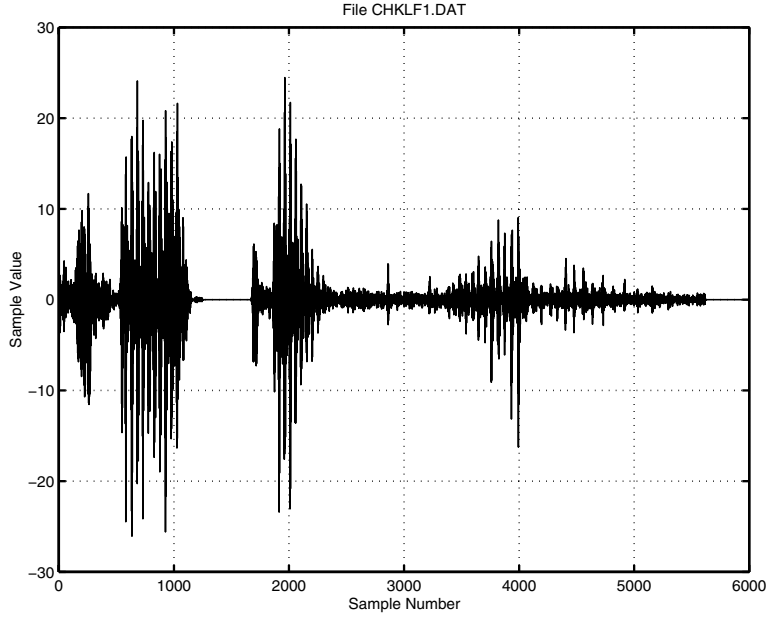


(b)

FIGURE 4.5 FIR lowpass filtering example. (a) Original WAVE3.DAT data file with 3 frequencies at 0.01, 0.02 and 0.4 (150 points). (b) After lowpass filter using the filter shown in Figure 4.2.



(a)



(b)

FIGURE 4.6 FIR highpass filtering example. (a) Original CHKL.DAT data file consisting of the author's words "chicken little" sampled at 8 kHz (6000 samples are shown). (b) After highpass filter using the filter shown in Figure 4.3.

FIRFILT program. This highpass result shows the high frequencies present in the "chi" and "ken" portions of the speech data. The CHKL.DAT data is analyzed in more detail in Section 5.10 of Chapter 5.

4.2.2 Integer FIR Filters

An *integer filter* can be designed by rounding the coefficients obtained from the McClellan-Park program. For best performance, the maximum coefficient in the FIR filter is set equal to the maximum integer value represented in the integer representation with B bits. In two's complement binary representation, values from -2^{B-1} to $2^{B-1}-1$ can be represented. For example, the 35-point lowpass filter shown in Figure 4.1 has a maximum value of 0.44541 at the center of the filter. The coefficients of this lowpass filter can be quantized using the following code segment:

```
for(i = 0 ; i < 35 ; i++)
    fir_int = round(127.0*fir_lpf35/0.44541);
```

Where the inline **round** function (defined in DSP.H) is used to round the floating-point values (**FIRLPF35** is defined in FILTER.CPP) to the nearest integer. The first 18 unique coefficients of this filter and the rounded values are as follows:

FIRLPF35 value	8-bit rounded value
-0.006849	-2
0.001949	1
0.013099	4
0.011007	3
-0.006610	-2
-0.013219	-4
0.006820	2
0.022924	7
0.000773	0
-0.031535	-9
-0.013848	-4
0.040546	12
0.038411	11
-0.047905	-14
-0.897390	-26
0.052856	15
0.312651	89
0.445415	127

The frequency response of this 8-bit integer filter is shown in Figure 4.7. Note that the minimum attenuation of this filter is not constant throughout the stopband. Thus, the

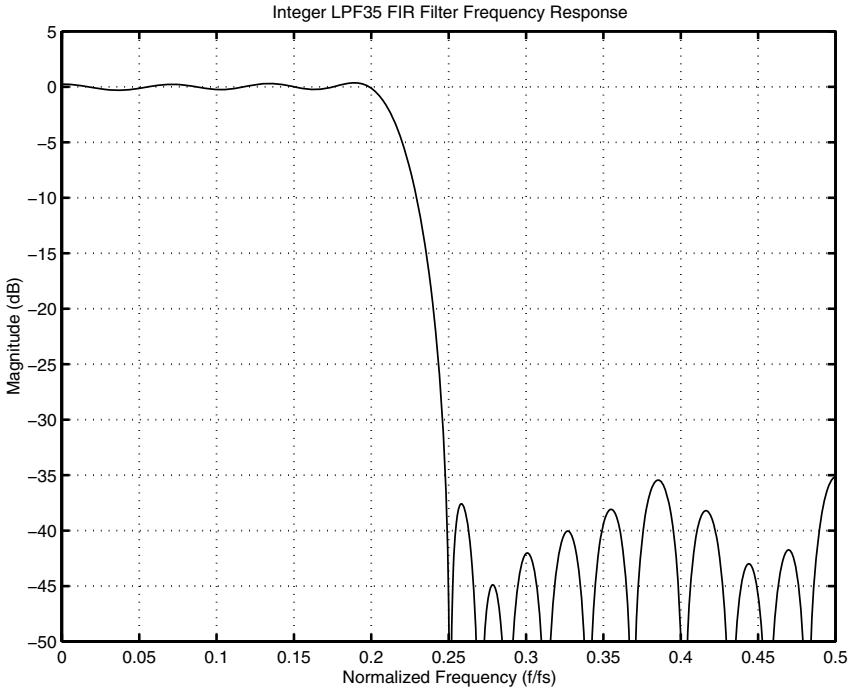


FIGURE 4.7 Frequency response of 35-point FIR lowpass filter after the coefficients have been quantized to 8 bits (maximum value = 127).

filter is no longer an equiripple filter, because of the quantization error in the coefficients. The overall minimum stopband attenuation is reduced to about 35 dB compared to the original 40.6 dB realized by the floating-point version [see Figure 4.2(b)]. A similar change in the passband ripple can also be discovered by plotting the passband in detail.

The program FIRITEST shown in Listing 4.3 implements direct convolution of an integer input and a set of integer coefficients to generate an integer result. For this example, the program will lowpass filter the integer CHKL.DAT data set.

The FIRITEST program works in a similar way to the FIRFILT program except that the product of the integer input and filter arrays is accumulated using a long integer (32 bits). In the above example, the accumulated results are divided by 285 because 127/285 is very close to the original max value of 0.44541. This scaling makes the result of the above program essentially identical to the result that would be obtained using the FIRFILT floating-point program. The differences between integer filters and floating-point filters are generally subtle except when the number of bits used to represent the filter is small (especially when less than 8 bits are used) or the required stopband attenuation is large (greater than 40 dB for 8-bit coefficients, for example).

```
//////////////////////////////////////////////////////////////////
//
// firitest.cpp - Filter vector of integers
//
//////////////////////////////////////////////////////////////////
#include "dsp.h”"
#include "disk.h"
#include "get.h"
#include "filter.h"
#include "fircoefs.h"

//////////////////////////////////////////////////////////////////
//
// int main()
//   Filters DSPFile containing integers using
//   templated FIRFilter class and generates filtered DSPFile.
//
//  Returns:
//   0 —- Success
//
//////////////////////////////////////////////////////////////////
int main()
{
    try
    {
     DSPFile dspfile;
     const int filtLen = ELEMENTS( FIRLPF35 );
     int intCoefs[filtLen];

     // Round the 35 point lowpass filter to 8 bits
     for( int i = 0; i < filtLen; i++ )
      intCoefs[i] = round( 127.0f * FIRLPF35[i] / 0.44541f );

     // Open the input file
     dspfile.openRead( "chkl.dat" );

     // Read data into integer vector
     Vector<int> vIn;
     dspfile.read( vIn );
     dspfile.close();

     // Create filter using integer coefficients
     FIRFilter<int> firiFilt( intCoefs, filtLen );
```

LISTING 4.3 Program **firitest** used to filter integer array using an integer array of coefficients. (*Continued*)

```
     // Filter file
     Vector<int> vOut = firiFilt.filter( vIn );

     // Scale output by precomputed constant
     for( i = 0; i < vOut.length(); i++ )
         vOut[i] = vOut[i] / 285;

     // Write filtered vector out to file
     dspfile.openWrite( "cifil.dat" );
     dspfile.write( vOut );
     dspfile.close();
    }
    catch( DSPException& e )
    {
     // Display exception
     cerr << e;
     return 1;
    }
    return 0;
}
```

LISTING 4.3 (*Continued*)

4.3 IIR FILTERS

Infinite impulse response filters are commonly realized recursively by feeding back a weighted sum of past output values and adding this to a weighted sum of the previous and current input values. In terms of the structure shown in Figure 4.1, IIR filters have nonzero values for some or all of the a_n values. The major advantage of IIR filters compared to FIR filters is that a given-order IIR filter can be made much more frequency selective than the same-order FIR filter. In other words, IIR filters are computationally efficient. The disadvantage of the recursive realization is that IIR filters are much more difficult to design and implement. Stability, roundoff noise, and sometimes phase nonlinearity must be considered carefully in all but the most trivial IIR filter designs.

The direct form IIR filter realization shown in Figure 4.1, though simple in appearance, can have severe response sensitivity problems because of coefficient quantization, especially as the order of the filter increases. To reduce these effects, the transfer function is usually decomposed into second-order sections and then realized either as parallel or cascade sections (see Chapter 1, Section 1.3). In Section 4.3.1 we describe an IIR filter design and implementation method based on cascade decomposition of a transfer function into second-order sections. The C++ language implementation given in Section 4.3.2 uses single-precision floating-point numbers in order to avoid coefficient quantization ef-

fects associated with fixed-point implementations that can cause instability and significant changes in the transfer function.

Figure 4.8 shows a block diagram of the cascaded second-order IIR filter implementation described in Section 4.3.2. This realization is known as a direct form II realization because unlike the structure shown in Figure 4.1, it has only two delay elements for each second-order section. This realization is canonic in the sense that the structure has the fewest adds (4), multiplies (4), and delay elements (2) for each second-order section. This realization should be the most efficient for a wide variety of general-purpose processors as well as many of the processors designed specifically for digital signal processing.

4.3.1 IIR Filter Design

IIR digital filters can be designed in many ways, but by far the most common IIR design method is the *bilinear transform.* This method relies on the existence of a known *s*-domain transfer function (or Laplace transform) of the filter to be designed. The *s*-domain filter coefficients are transformed into equivalent *z*-domain coefficients for use in an IIR digital filter. This might seem like a problem, since *s*-domain transfer functions are just as hard to determine as *z*-domain transfer functions. Fortunately, Laplace transform methods and *s*-domain transfer functions were developed many years ago for designing analog filters as well as for modeling mechanical and even biological systems.

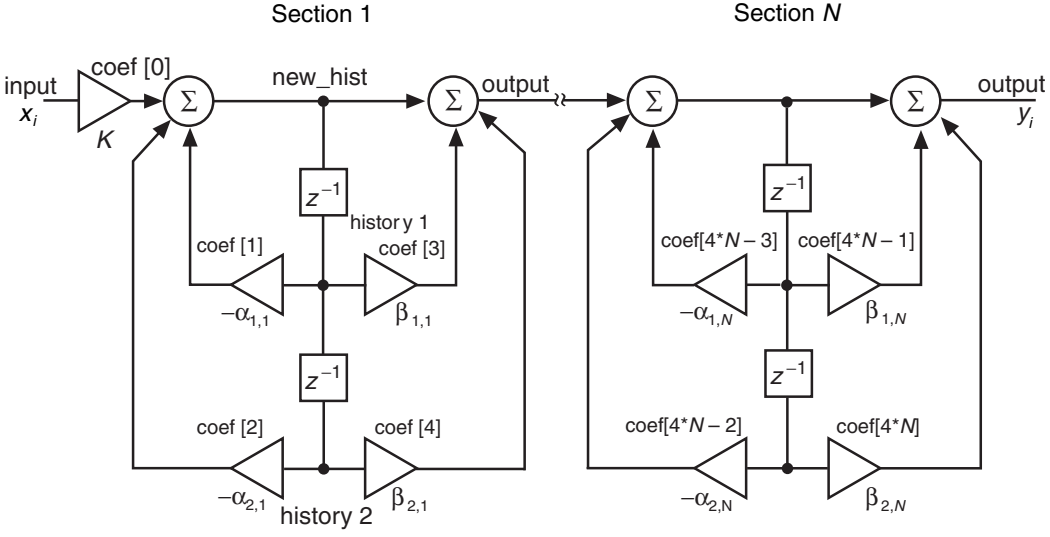


FIGURE 4.8 Direct form II realization of *N* second order cascaded IIR sections as implemented by **IIRFilter.** The variables **coef[ ], new-hist, history 1,** and **history 2** refer to the variable names used by functions in **IIRFilter.**

Thus, many tables of *s*-domain filter coefficients are available for almost any type of filter function (see the References for a few examples). Also, computer programs are available to generate coefficients for many of the common filter types (see the books by Jong, Antoniou, and Stearns and David, or one of the many filter design packages available for personal computers). Because of the vast array of available filter tables, the large number of filter types and because the design and selection of a filter requires careful examination of all the requirements (passband ripple, stopband attenuation, as well as phase response in some cases), the subject of *s*-domain filter design will not be covered in this book. Instead, this section describes the bilinear transform and introduces a program to perform the bilinear transform on a set of *s*-domain coefficients (program IIRDEZN.CPP shown in Listing 4.4). The *s*-domain coefficients are assumed to be factored into a series of second-order sections for a cascade filter implementation. Most of the tables of *s*-domain filter coefficients are provided in this format or can easily be converted. The *z*-domain coefficients from the bilinear transform can then be used directly by the **IIRFilter** class introduced in Section 4.2.2.

```
//////////////////////////////////////////////////////////////////////////
//
//iirdezn.cpp - Design IIR filter coefficients
//
//////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "filter.h"

//////////////////////////////////////////////////////////////////////////
//
//Constants
//
//////////////////////////////////////////////////////////////////////////

// Size of magnitude vector to write
const int MAG_LENGTH = 501;

//////////////////////////////////////////////////////////////////////////
//
//prewarp( Vector<double>& w, double fc, double fs )
//  Prewarps coefficents of a numerator or denominator.
//
//////////////////////////////////////////////////////////////////////////
```

LISTING 4.4 Program IIRDEZN used to determine the *z*-domain of an IIR filter from a set of *s*-domain filter coefficients. (*Continued*)

```
void prewarp( Vector<double>& w, double fc, double fs )
{
    // Validate parameters
    if( w.length() != 3 )
        throw DSPParamException("Prewarp vector must contain 3 elements");

    double wp = 2.0 * fs * tan( PI * fc / fs );

    if( wp == 0.0 )
        throw DSPMathException( "Divide by zero" );

    w[1] = w[1] / wp;
    w[2] = w[2] / ( wp * wp );
}

////////////////////////////////////////////////////////////////////////////
//
// bilinear( ... )
//   Compute the bilinear tranform of the numerator and
//   denominator coefficients and return four z transform
//   coeffiecients for Direct Form II realization using
//   the IIR filter program.
//
////////////////////////////////////////////////////////////////////////////
void bilinear(
    // Numerator coefficients
    const Vector<double>& a,
    // Denominator coefficients
    const Vector<double>& b,
    // Overall gain factor
    double& k,
    // Sampling rate
    double fs,
    // 4 IIR coefficients
    float *coef )
{
    // Validate parameters
    if( coef == NULL )
        throw DSPParamException("NULL filter coefficients");
    if( a.length() != 3 || b.length() != 3 )
        throw DSPParamException("Invalid coefficient vector length");

    // Alpha denominator
    double ad = 4.0f * a[2] * fs * fs + 2.0f * a[1] * fs + a[0];
```

LISTING 4.4 (*Continued*)

```
    // Beta denominator
    double bd = 4.0f * b[2] * fs * fs + 2.0f * b[1] * fs + b[0];

    // Update gain constant for this section
    if( bd == (double)0.0f )
        throw DSPMathException( "Divide by zero" );

    k *= ad / bd;

    // Bilinear transform the coefficients

    // Beta1
    coef[0] = (float)( ( 2.0f*b[0] - 8.0f*b[2]*fs*fs ) / bd );
    // Beta2
    coef[1] = (float)( ( 4.0f*b[2] * fs*fs - 2.0f*b[1]*\fs + b[0] ) / bd );
    // Alpha1
    coef[2] = (float)( ( 2.0f*a[0] - 8.0f*a[2]*fs * fs ) / ad );
    // Alpha2
    coef[3] = (float)( ( 4.0*a[2]*fs*fs - 2.0f*a[1]*fs + a[0] ) / ad );
}
////////////////////////////////////////////////////////////////////////////
//
// Vector<double> magnitude( const IIRFilter<float>& iir )
//   Finds the magnitude of an IIR filter.
//
//  Returns:
//   Double vector sized MAG_LENGTH with magnitude 0 to 0.5
//
////////////////////////////////////////////////////////////////////////////
Vector<double> magnitude( const IIRFilter<float>& iir )
{
    Vector<double> vMag( MAG_LENGTH );

    // Loop through all the frequencies
    for( int i = 0; i < MAG_LENGTH; i++ )
    {
        double f = (double) i / ( 2.0f * ( MAG_LENGTH - 1 ) );
        double arg = 2.0 * PI * f;
        double z1r = cos( arg );
        double z1i = -sin( arg );
        double z2r = cos( 2.0f * arg );
        double z2i = -sin( 2.0 * arg );

        // Coefficient pointer
        const float *coef = iir.getCoefficients( iir.length() );
```

LISTING 4.4 (*Continued*)

```
        // Overall K first
        vMag[i] = *coef++;

        // Loop through all sections: numerator and denominator
        for( int j = 0; j < iir.length(); j++ )
        {
            // Get the four coefficients for each section
            double beta1 = *coef++;
            double beta2 = *coef++;
            double alpha1 = *coef++;
            double alpha2 = *coef++;

            // Denominator
            double d_real = 1.0f + beta1 * z1r + beta2 * z2r;
            double d_imag = beta1 * z1i + beta2 * z2i;
            vMag[i] = vMag[i] / ( d_real * d_real + d_imag * d_imag );

            // Numerator
            double n_real = 1 + alpha1*z1r + alpha2*z2r;
            double n_imag = alpha1 * z1i + alpha2 * z2i;
            double real = n_real * d_real + n_imag * d_imag;
            double imag = n_imag * d_real - n_real * d_imag;
            vMag[i] = vMag[i] * sqrt( real * real + imag * imag );
        }
    }
    return vMag;
}

////////////////////////////////////////////////////////////////////////////
//
// int main()
//   Design IIR filter coefficients from analog prototype
//   coefficients and determine magnitude response.
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        // For formatting output
        char szFmt[300];
```

LISTING 4.4 (*Continued*)

```
// Get user parameters
double fs = 0.0f;
getInput( "Enter sampling rate", fs, 0.0, 1.e9 );
double fc = 0.0f;
getInput( "Enter cutoff frequency", fc, 0.0, ( fs / 2.0 ) );
int sections = 0;
getInput( "Enter number of sections", sections, 1, 20 );

// Allocate coefficient array (used to create filter)
int length = ( 4 * sections ) + 1;
float *coefArray = new float[length];
if( coefArray == NULL )
    throw DSPMemoryException( "Coefficient allocation failed" );
float *coef = coefArray + 1;

// Start gain at 1.0
double k = 1.0;

for( int i = 1; i <= sections; i++ )
{
    cout << "Enter section #" << i << " normalized coeffs:\n";

    // Numerators
    Vector<double> a( 3 );
    getInput( "Numerator A0", a[0], -1.e6, 1.e6 );
    getInput( "Numerator A1", a[1], -1.e6, 1.e6 );
    getInput( "Numerator A2", a[2], -1.e6, 1.e6 );

    prewarp( a, fc, fs );

    // Denominators
    Vector<double> b( 3 );
    getInput( "Denominator B0", b[0], -1.e6, 1.e6 );
    getInput( "Denominator B1", b[1], -1.e6, 1.e6 );
    getInput( "Denominator B2", b[2], -1.e6, 1.e6 );

    prewarp( b, fc, fs );
    bilinear( a, b, k, fs, coef );
    coef += 4;
}

// Store gain constant
coef = coefArray;
coef[0] = (float)k;
```

LISTING 4.4 (*Continued*)

```
// Display coefficients
cout << "Z domain IIR coefficients are as follows:\n";
for( i = 0; i < length; i++ )
{
    sprintf( szFmt, "C[%d] = %15.10f\n", i, coef[i] );
    cout << szFmt;
}

// Determine magnitude transfer function
IIRFilter<float> iir( coef, length, sections );
Vector<double>vMag = magnitude( iir );

// Open output magnitude file
DSPFile dspfile;
String strName;
String strTrailer;

do getInput( "Enter output file name", strName );
while( strName.isEmpty() );
dspfile.openWrite( strName );
dspfile.write( vMag );

// Determine log of magnitude in dB and write out
for( i = 0; i < MAG_LENGTH; i++ )
{
    if( vMag[i] < 1.e-4 )
        vMag[i] = -80.0;
    else
        vMag[i] = 20.0 * log10( vMag[i] );
}
dspfile.write( vMag );

// Make descriptive trailer and write to file
strTrailer = "Response of IIR filter, mag and 20*log(mag)\n";
sprintf( szFmt, "Fc = %g Fs = %g\n", fc, fs);
strTrailer += szFmt;
strTrailer += "Z domain IIR coefficients are as follows:\n";
for( i = 0; i < ( 4 * sections + 1 ); i++ )
{
    sprintf( szFmt, "\nC[%d] = %15.10f", i, coefArray[i] );
    strTrailer += szFmt;
}
dspfile.setTrailer( strTrailer );
dspfile.close();
```

LISTING 4.4 (*Continued*)

```
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 4.4 (*Continued*)

The form of the *z*-domain transfer function that is implemented by the **IIRFilter** member function is the product of *N* biquadratic functions as follows:

$$H(z) = K\prod_{n=1}^{N}\frac{1+\alpha_{1,n}z^{-1}+\alpha_{2,n}z^{-2}}{1+\beta_{1,n}z^{-1}+\beta_{1,n}z^{-2}} \tag{4.5}$$

Thus, IIR filter design consists of determining the coefficients of the *N* second-order sections of the above form (sometimes called *biquads*) that are required to realize a particular transfer function. The four real coefficients ($\alpha_{1,n}$, $\alpha_{2,n}$, $ß_{1,n}$, $ß_{2,n}$) of each biquad determine the transfer function of the second-order section and the location of two poles and two zeroes on the *z* plane. The order of $H(z)$ is $2N$ so that if an odd order $H(z)$ is desired, a first-order section must be included such that $\alpha_{2,n}$ and $ß_{2,n}$ are both zero. The gain constant, *K,* in the above equation determines the overall passband gain of the entire filter. In the floating-point implementation described in Section 4.2.2, this coefficient is used to scale the input amplitude. The *s*-domain transfer function of the corresponding analog filter is as follows:

$$H(s) = K\prod_{n=1}^{N}\frac{a_{2,n}s^2+a_{1,n}s+a_{0,n}}{b_{2,n}s^2+b_{1,n}s+b_{0,n}} \tag{4.6}$$

The *s*-domain coefficients are normally provided in tables in a form that gives the values of the six coefficients in the above equation for a lowpass filter with a cutoff frequency (the frequency where the passband ends) at 1 radian per second. In other words, if *s* is set equal to 1*j,* then the magnitude of the filter transfer function will be less than unity by an amount equal to the passband ripple of the design (3 dB for Butterworth designs, for example). The normalized lowpass coefficients (often called lowpass prototype coefficients) can be used to determine the response of a highpass filter by substituting *s* in the above equation for 1/*s*' (where *s*' is used in the *s*-domain transfer function of the resulting highpass filter). The bandpass transformation is a bit more complicated and results in a doubling of the order of the filter. The resulting transformed lowpass sections (now 4th order) must each be factored into a pair of new biquadratic sections. The reader is re-

ferred to the references for a detailed discussion concerning this procedure (see Van Valkenburg for a description of the Geffe algorithm, for example).

The bilinear transform can now be applied to the normalized *s*-domain coefficients for the desired lowpass, highpass, or bandpass filter. First, the coefficients must be transformed to the correct cutoff frequency by using the bilinear transform prewarping relationship as follows:

$$C_W = \frac{f_s}{\pi} \tan\left(\frac{\pi f}{f_s}\right) \tag{4.7}$$

where C_W is the prewarping constant, f is the desired cutoff frequency (in Hertz) of the overall filter, and f_s is the sampling rate in Hertz. Note that the desired cutoff frequency must be less than half of the sampling frequency in order to be able to realize the filter. The prewarping constant is then used to change the cutoff frequency of all the second-order sections. All second-order coefficients ($a_{2,n}$ and $b_{2,n}$) are divided by $C_W{}^2$ and the first-order coefficients ($a_{1,n}$ and $b_{1,n}$) are divided by C_w. The function **prewarp** in program IIRDEZN.CPP contains the required equations to prewarp each second-order section according to Equation (4.7).

Now that the *s*-domain poles and zeroes are set in the correct position for the bilinear transform at the desired sampling frequency, the bilinear transform can be applied to each second-order section. In each case *s* (or *s*' in the bandpass or highpass case), is substituted as follows:

$$s = 2f_s \frac{1 - z^{-1}}{1 + z^{-1}} \tag{4.8}$$

This results in a *z*-domain transfer function of the same order as the *s*-domain transfer function with the four coefficients for each biquad section as follows:

$$\alpha_{1,n} = \frac{-8a_{2,n}f_s^2 + 2a_{0,n}}{4a_{2,n}f_s^2 + 2a_{1,n}f_s + a_{0,n}} \tag{4.9}$$

$$\alpha_{2,n} = \frac{4a_{2,n}f_s^2 - 2a_{1,n}f_s + a_{0,n}}{4a_{2,n}f_s^2 + 2a_{1,n}f_s + a_{0,n}} \tag{4.10}$$

$$\beta_{1,n} = \frac{-8b_{2,n}f_s^2 + 2b_{0,n}}{4b_{2,n}f_s^2 + 2b_{1,n}f_s + b_{0,n}} \tag{4.11}$$

$$\beta_{2,n} = \frac{4a_{2,n}f_s^2 - 2a_{1,n}f_s + a_{0,n}}{4b_{2,n}f_s^2 + 2b_{1,n}f_s + b_{0,n}} \tag{4.12}$$

The overall gain constant, *K,* is given by the following product:

$$K = \prod_{n=1}^{N} \frac{4b_{2,n}f_s^2 + b_{1,n}f_s + b_{0,n}}{4a_{2,n}f_s^2 + a_{1,n}f_s + a_{0,n}} \tag{4.13}$$

Equations 4.9 through 4.13 are implemented in the function **bilinear** in the IIRDEZN.CPP program and are used to transform each section's *s*-domain coefficients into the *z*-domain coefficients required for the **IIRFilter** function.

Some of the more important properties of the bilinear transform not mentioned so far are as foIlows:

1. Poles and zeroes in the left half *s*-plane are mapped to poles and zeroes inside the *z*-plane unit circle. The imaginary *j*;gv-axis of the *s*-plane is mapped onto the unit circle of the *z*-plane. Thus, stable *s*-domain transfer functions map into stable *z*-domain transfer functions. However, the *z*-domain coefficients must be represented with enough precision to give a stable digital filter. For this reason, high-order digital filters with narrow bandwidths must often be implemented using floating-point arithmetic or using fixed-point arithmetic with a large number of bits.
2. Both the numerator and denominator of each *z*-domain second-order section are second order. It is unusual for any of the four *z*-domain coefficients to be zero.
3. All finite analog frequencies in the *s*-domain are mapped into frequencies that are bounded by $f_s/2$. This is a result of the prewarping of the *s*-domain poles and zeroes. For example, the *s*-domain zero at infinity of lowpass and bandpass filters is mapped to $f_s/2$ in the digital frequency response.
4. The *z*-domain transfer function obtained from the bilinear transform is at best a good approximation of the analog frequency response. As the input frequency increases toward $f_s/2$, the digital filter response becomes less representative of the analog frequency response. For lowpass Butterworth and Chebychev digital filters, this property enhances the stopband attenuation because the bilinear transform places a zero at $f_s/2$.

The use of the IIRDEZN program will be illustrated using a lowpass filter with the same specifications as was used in the FIR filter design example given in Section 4.2. The only difference is that in the IIR filter specification, linear phase response is not required. Thus, the passband is 0 to $0.2f_s$ and the stopband is 0.25 f_s to $0.5f_s$. The passband ripple must be less than 0.5 dB and the stopband attenuation must be greater than 40 dB. Because elliptic filters (also called *Cauer filters*) generally give the smallest transition bandwidth for a given order, an elliptic design will be used. After referring to the many elliptic filter tables, it is determined that a fifth-order elliptic filter will meet the specifications. The elliptic filter tables in the book by Zverev (see References) give an entry for a filter with a 0.28 dB passband ripple and 40.19 dB stopband attenuation as follows:

$\Omega_s = 1.3250$	(stopband start of normalized prototype)
$\sigma_0 = -0.5401$	(first-order real pole)
$\sigma_1 = 0.08058$	(real part of first biquad section)
$\sigma_3 = 0.32410$	(real part of second biquad section)
$\Omega_1 = 1.0277$	(imaginary part of first biquad section)

$\Omega_2 = 1.9881$ (first zero on imaginary axis)
$\Omega_3 = 0.7617$ (imaginary part of second biquad section)
$\Omega_4 = 1.3693$ (second zero on imaginary axis)

As shown above, the tables in Zverev give the pole and zero locations (real and imaginary coordinates) of each biquad section. The two second-order sections each form a conjugate pole pair, and the first-order section has a single pole on the real axis. Figure 4.9 shows the locations of the 5 poles and 4 zeroes on the complex s-plane. By expanding the complex pole pairs, the following general form for the s-domain domain transfer function of a fifth order filter in terms of the above variables is obtained:

$$\frac{\sigma_0}{s+\sigma_0}\,\frac{(\sigma_1^2+\Omega_1^2)\,(s^2+\Omega_2^2)}{\Omega_2^2\,(s^2+2\sigma_1 s+\sigma_1^2+\Omega_1^2)}\,\frac{(\sigma_3^2+\Omega_3^2)\,(s^2+\Omega_4^2)}{\Omega_4^2\,(s^2+2\sigma_3 s+\sigma_3^2+\Omega_3^2)}$$

This results in a transfer function of the lowpass prototype as follows:

$$\frac{0.5401}{s+0.5401}\,\frac{0.2689\,s^2+1.0627}{s^2+0.1612s+1.0627}\,\frac{0.3655\,s^2+0.6852}{s^2+0.6482s+0.6852}$$

The above coefficients can now be used with the IIRDEZN program. Each numerator and denominator coefficient is entered for each section. The following dialogue illustrates the use of IIRDEZN for the above filter with a cutoff frequency of $0.2f_s$:

```
Enter sampling rate [0...1e + 009] : 1

Enter filter cutoff frequency [0...0.5] : 0.2

Enter number of sections [1...20] : 3

Enter section #1 normalized coefficients—
Enter numerator A0 [-1e+006...1e+006] : 0.5401

Enter numerator A1 [-1e+006...1e+006] : 0

Enter numerator A2 [-1e+006...1e+006] : 0

Enter denominator B0 [-1e+006...1e+006] : 0.5401

Enter denominator B1 [-1e+006...1e+006] : 1

Enter denominator B2 [-1e+006...1e+006] : 0
Enter section #2 normalized coefficients—
Enter numerator A0 [-1e+006...1e+006] : 1.0626604

Enter numerator A1 [-1e+006...1e+006] : 0

Enter numerator A2 [-1e+006...1e+006] : 0.268855

Enter denominator B0 [-1e+006...1e+006] : 1.0626604
```

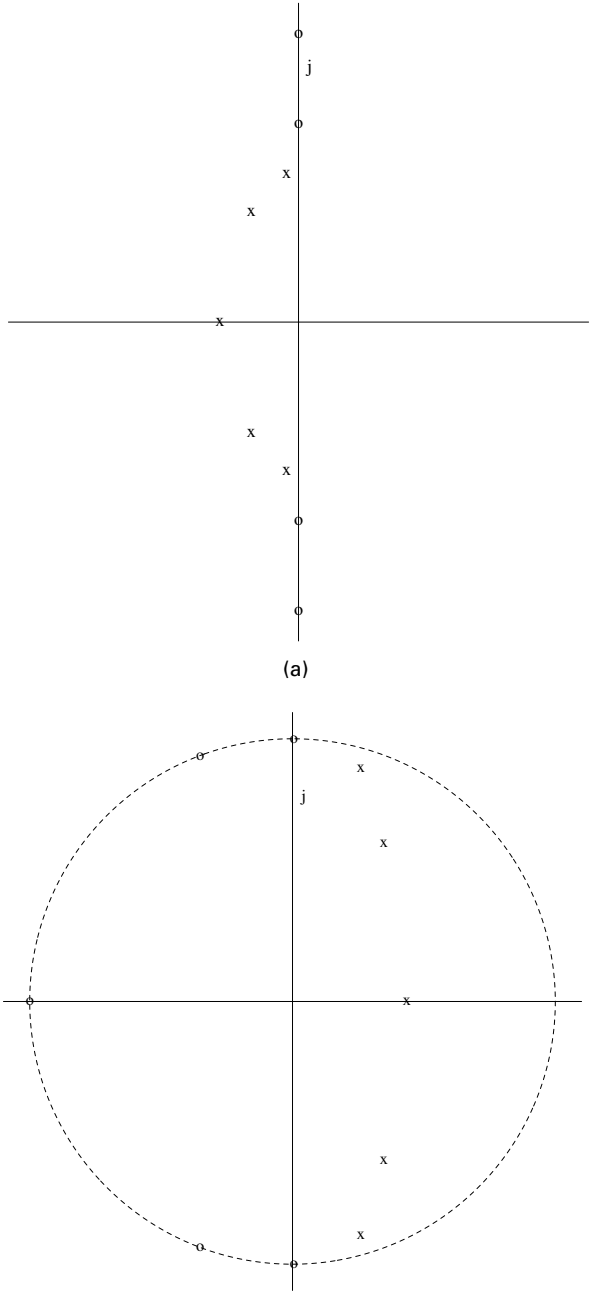


FIGURE 4.9 Pole-zero plot of fifth order elliptic IIR lowpass filter. (a) *s*-plane representation of analog prototype. (b) *z*-plane representation of lowpass digital filter with cut-off frequency at $0.2f_s$. In each case, poles are indicated with "x" and zeroes with "o."

```
Enter denominator B1 [-1e+006...1e+006] : 0.16116

Enter denominator B2 [-1e+006...1e+006] : 1

Enter section #3 normalized coefficients—
Enter numerator A0 [-1e+006...1e+006] : 0.685228

Enter numerator A1 [-1e+006...1e+006] : 0

Enter numerator A2 [-1e+006...1e+006] : 0.36546

Enter denominator B0 [-1e+006...1e+006] : 0.685228

Enter denominator B1 [-1e+006...1e+006] : 0.64820

Enter denominator B2 [-1e+006...1e+006] : 1

Z domain IIR coefficients are as follows:

C[0] =   0.0552961603
C[1] =   0.5636369586
C[2] =  -0.4363630712
C[3] =   2.0000000000
C[4] =   1.0000000000
C[5] =  -0.5233039260
C[6] =   0.8604439497
C[7] =   0.7039934993
C[8] =   1.0000000000
C[9] =  -0.6965782046
C[10] =  0.4860509932
C[11] = -0.0103216320
C[12] =  1.0000000000

Enter file name: LPF5E.MAG
```

The above dialogue requires a great deal of data entry by the user, thereby making the IIRDEZN program difficult to use for high-order filters. To alleviate this problem, the input to the program can be stored in a disk file. The file LPF5E on the accompanying disk contains the input data shown above. The contents of the LPF5E disk file can then be directed to the input of IIRDEZN by the following operating system redirection (this works on most systems):

```
IIRDEZN < LPF5E
```

The result of the IIRDEZN program is exactly the same as typing in each of the user inputs. The DSP data file LPF5E.MAG contains the frequency response of the IIR filter in the DSP data format suitable for plotting using the WINPLOT program. The magnitude of the IIR filter is calculated from the z-domain coefficients in the function **magnitude**

(shown in Listing 4.4). Figure 4.10 shows the two records in this file that gives the linear magnitude response (where the passband ripple can be seen) and the magnitude in decibels (which shows the stopband attenuation).

The z-domain coefficients generated by the IIRDEZN program are given in the following order: K, $ß_{1,1}$, $ß_{2,1}$, $\alpha_{1,1}$, $\alpha_{2,1}$, $ß_{1,2}$, $ß_{2,2}$, $\alpha_{1,2}$, $\alpha_{2,2}$, $ß_{1,3}$, $ß_{2,3}$, $\alpha_{1,3}$, $\alpha_{2,3}$. This order of the coefficients is used directly by the **IIRFilter** function discussed in Section 4.2.2. The resulting z-domain transfer function is as follows:

$$\frac{0.0553(1+z^{-1})}{1-0.436z^{-1}}\frac{1+0.704z^{-1}+z^{-2}}{1-0.523z^{-1}-0.86z^{-2}}\frac{1-0.0103z^{-1}+z^{-2}}{1-0.696z^{-1}-0.486z^{-2}}$$

Note that the first-order section in the s-domain resulted in a second-order section in the z-domain. This is because the IIRDZEN program implemented Equations 4.9 to 4.12 which were developed for second-order sections. The first biquad in the above transfer function (which results from the first-order s-term) can be reduced to first order by factoring a $(1 + z^{-1})$ term from the numerator and the denominator. This factoring would result in some computational savings in the implementation of the odd-order section, since the $\alpha_{2,1}$ and $ß_{2,1}$ terms are zero. This procedure is required only if the filter order is odd and if the implementation can take advantage of the reduced number of multiplies. The final reduced form of the fifth-order z-domain transfer function is as follows:

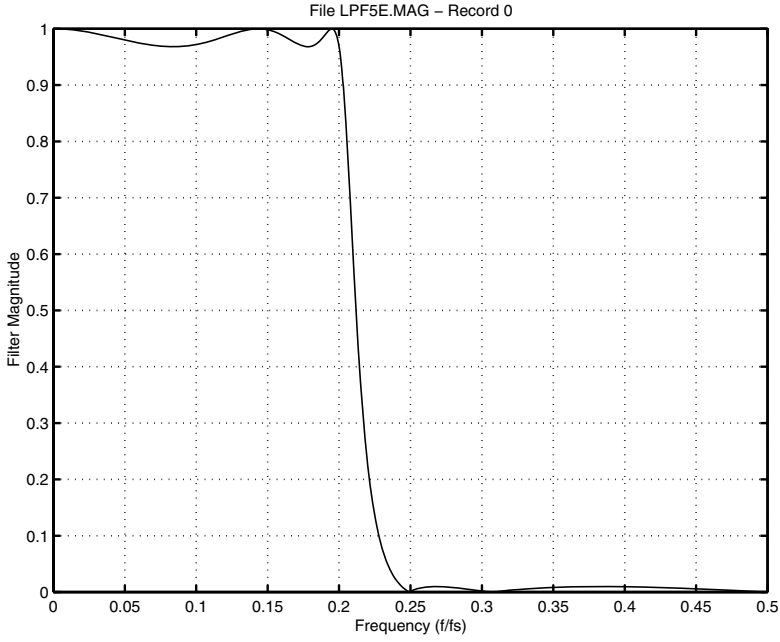
$$\frac{0.0553(1+z^{-1})}{1-0.436z^{-1}}\frac{1+0.704z^{-1}+z^{-2}}{1-0.523z^{-1}-0.86z^{-2}}\frac{1-0.0103z^{-1}+z^{-2}}{1-0.696z^{-1}-0.486z^{-2}}$$

The pole and zero locations of this z-domain transfer function are shown in Figure 4.9(b). Note that the five zeroes at $\pm 0.2492f_s$, $\pm 0.3072f_s$, and $0.5f_s$ appear on the unit circle. The five poles are progressively closer to the unit circle as the cut-off frequency (at $0.2f_s$) is approached but will result in a stable transfer function, since they are all inside the unit circle.

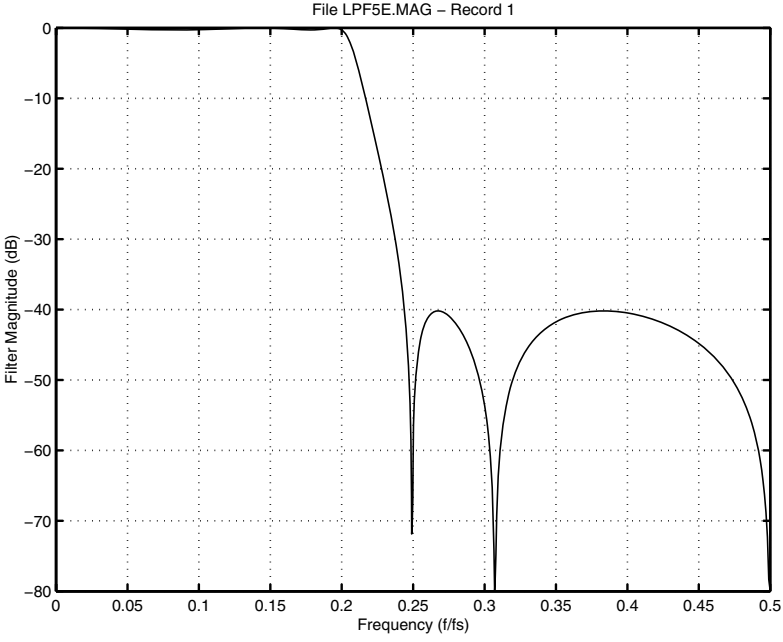
4.3.2 IIR Filter Function

The class **IIRFilter** (shown in Listing 4.5) implements the direct form II cascade filter structure illustrated in Figure 4.8. Any number of cascaded second-order sections can be implemented with one overall input (x_i) and one overall output (y_i). The **Filter** structure for the fifth-order elliptic lowpass filter described in the previous section is defined as follows:

```
////////////////////////////////////////////////////////////////
//
// IIR lowpass 3 section (5th order) elliptic filter
// with 0.28 dB passband ripple and 40 dB stopband attenuation.
// The cutoff frequency is 0.25*fs.
//
////////////////////////////////////////////////////////////////
```



(a)



(b)

FIGURE 4.10 Lowpass fifth-order elliptic IIR filter frequency response. (a) Linear magnitude versus frequency. (b) Log magnitude in decibels versus frequency.

```
float IIRLPF5[] =
{
     0.0552961603F,
    -0.4363630712F, 0.0000000000F,  1.0000000000F, 0.0000000000F,
    -0.5233039260F, 0.8604439497F,  0.7039934993F, 1.0000000000F,
    -0.6965782046F, 0.4860509932F, -0.0103216320F, 1.0000000000F
};
IIRFilter<float> IIRLPF( IIRLPF5, ELEMENTS( IIRHPF6 ), 3 );
```

The number of sections (stored in **m_length**) required for this filter is three because the first-order section is implemented in the same way as the second-order sections except that the second-order terms (the third and fifth coefficients) are zero. The coefficients and structure shown above were obtained using the IIRDEZN program and are contained in the file FILTER.CPP. The definition of this filter is therefore global to any module which uses FILTER.CPP.

The **IIRFilter** member function filters the floating-point input sequence on a sample-by-sample basis such that one output sample is returned each time **IIRFilter** is invoked. The history pointer in the **Filter** class is used to store the two history values required for each second-order section. The history data is allocated during the first use of **IIRFilter** or whenever the history pointer is **NULL.** The initial condition of the history variables is set to zero. The coefficients of the filter are stored with the overall gain constant (*K*) first, followed by the denominator coefficients that form the poles and the numerator coefficients that form the zeroes for each section. The input sample is first scaled by the *K* value, and then each second-order section is implemented. The four lines of code in the **filter** function used to implement each second order section is as follows:

```
// Poles
output = output - ( hist1 * ( *ptrCoef++ ) );
Type newHist = output - ( hist2 * ( *ptrCoef++ ) );
// Zeros
output = newHist + ( hist1 * ( *ptrCoef++ ) );
output = output + ( hist2 * ( *ptrCoef++ ) );
```

The **hist1** and **hist2** variables are the current history associated with the section and should be stored in floating-point registers (if available) for highest efficiency. The above code forms the new history value (the portion of the output that depends on the past outputs) in the variable **newHist** to be stored in the history array for use by the next call to **IIRFilter.** The history array values are then updated as follows:

```
*ptrHist2++ = *ptrHist1;
*ptrHist1++ = newHist;
ptrHist1++;
ptrHist2++;
```

```
//////////////////////////////////////////////////////////////////////
//
// Class IIRFilter -- Implements abstract base class filter
//
//////////////////////////////////////////////////////////////////////
template <class Type>
class IIRFilter : public Filter<Type>
{
public:
    IIRFilter( const Type *coef, int length, int sections )
    {
     // Validate parameters
     if( length <= 0 )
       throw DSPParamException( "No filter data" );

     if( coef == NULL )
       throw DSPParamException( "Null filter coefficients" );

     if( ( sections * 4 ) + 1 != length )
       throw DSPParamException( "Coefficients do not match" );

     // Allocate memory for filter coefficients and history
     m_coef = new Type[length];
     if( m_coef == NULL )
       throw DSPMemoryException();

     m_hist = new Type[2 * sections];
     if( m_hist == NULL )
       throw DSPMemoryException();

     // Length of IIR filter is number of sections
     m_length = sections;

     // Copy coefficients
     Type *pc = m_coef;
     for( int i = 0; i < length; i++ )
       *pc++ = *coef++;
     reset();
    }

    // Implementation of abstract base class function (see Filter)
    virtual Type filterElement( const Type input )
    {
     // Coefficient pointer
```

LISTING 4.5 Class **IIRFilter** used to implement an **IIR** filter from a set of the *z*-domain coefficients. (*Continued*)

```
     Type *ptrCoef = m_coef;
     // History pointers
     Type *ptrHist1 = m_hist;
     Type *ptrHist2 = m_hist + 1;

     // Form output accumulation
     Type output = (Type)(input * ( *ptrCoef++ ) );
     for( int i = 0; i < m_length; i++ )
     {
       // Save history values
       Type hist1 = *ptrHist1;
       Type hist2 = *ptrHist2;

       // Poles
       output = output - ( hist1 * ( *ptrCoef++ ) );
       Type newHist = output - ( hist2 * ( *ptrCoef++ ) );

       // Zeros
       output = newHist + ( hist1 * ( *ptrCoef++ ) );
       output = output + ( hist2 * ( *ptrCoef++ ) );

       *ptrHist2++ = *ptrHist1;
       *ptrHist1++ = newHist;
       ptrHist1++;
       ptrHist2++;
     }
     return output;
    }

    // Implementation of abstract base class function (see Filter)
    virtual void reset()
    {
     Type *ph = m_hist;
     for( int i = 0; i < 2 * m_length; i++ )
       *ph++ = (Type)0;
    }

private:
    // No default constructor
    IIRFilter();
};
```

LISTING 4.5 (*Continued*)

This results in the oldest history value (***ptrHist2**) being lost and updated with the more recent ***ptrHist1** value. The **newHist** value replaces the old ***ptrHist1** value for use by the next call to **IIRFilter.** Both history pointers are incremented twice to point to the next pair of history values to be used by the next second-order section.

Listing 4.6 shows the program IIRFILT, which illustrates the use of the **IIRFilter** function. IIRFILT filters the records of a DSP data file by one of two digital IIR filters. The user is prompted for the input and output file names and choice of filters. The **IIRFilter** class is then invoked using the following code:

```
// Get record
    dspfileIn.read( vIn );
// Write filtered record
    dspfileOut.write( filter->filter( vIn ) );
// Reset history between records
    filter->reset();
```

The above code filters each record "in-place," which can be done because all of the history is stored in the **Filter** structure and each signal array element is only accessed once. After filtering each record of the input file, the **signal** and **history** arrays are reset to zero by the **reset** member function. This reset of the arrays ensures that the history from one record is not used with the next record.

The first filter (selection 0) is the fifth-order lowpass elliptic described in Section 4.3.1. The second filter choice (selection 1) is a sixth-order highpass Chebyshev filter with a passband from 0.3 to $0.5f_s$ and a passband ripple of 1 dB. The filter was designed using program IIRDEZN and the tables found in the book by Johnson et al. (with no conversion required). The input *s*-domain coefficients for this highpass filter are stored in file HPF6C. The *z*-domain coefficients can be obtained with the IIRDEZN program as follows:

```
IRRDEZN < HPF6C
```

The magnitude response of this highpass filter is shown in Figure 4.11 in the same format as Figure 4.10 for the lowpass filter. Note that the sixth-order Chebyshev filter gives a wider transition band than the highpass 35-point FIR filter (described in Section 4.2) but generally has a larger attenuation far from the passband.

The following dialogue shows the use of the IIRFILT program to filter the WAVE3.DAT data file using the fifth-order lowpass IIR filter (selection 0):

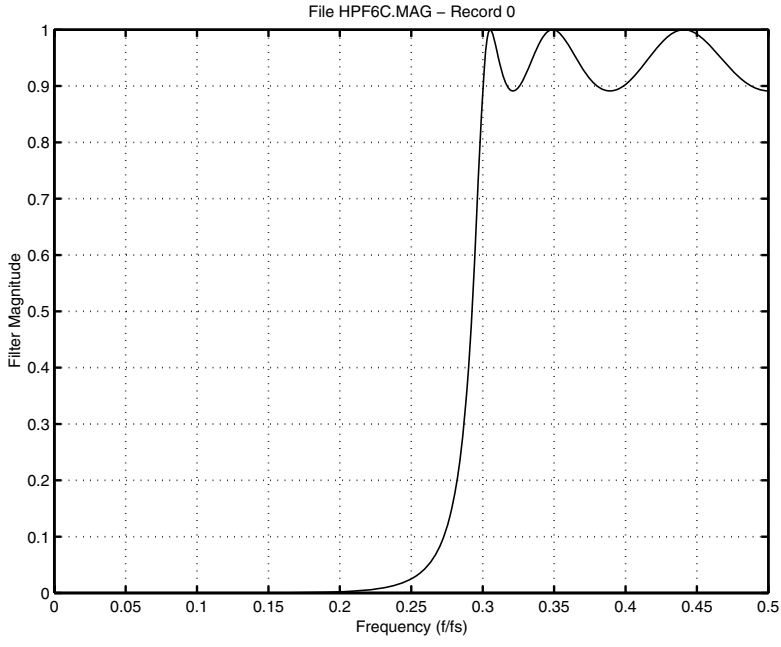
```
Enter input file name: WAVE3.DAT

File trailer:

Signal of length 150 equal to the sum of 3
cosine waves at the following frequencies:
```

```
//////////////////////////////////////////////////////////////////////
//
// iirfilt.cpp - Filter records using IIR Filters
//
//////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "filter.h"
#include "iircoefs.h"

//////////////////////////////////////////////////////////////////////
//
// int main()
//   Filters DSPFile records using a infinite-impulse-response
//   filter and generate a new DSPFile containing the filtered data.
//
//  Returns:
//   0 -- Success
//
//////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfileIn;
        DSPFile dspfileOut;
        String strName;
        String strTrailer;

        // Open the input file
        do getInput( "Enter file to filter", strName );
        while(strName.isEmpty() || dspfileIn.isFound(strName) == false );
        dspfileIn.openRead( strName );
        dspfileIn.getTrailer( strTrailer );

        // Print trailer
        if( strTrailer.isEmpty() == false )
            cout << "File trailer:\n" << strTrailer << endl;

        // Get type of filter to use of several defined in filter.h
        int typeFilter = 0;
        getInput(
            "Enter type of filter ( 0 = LPF, 1 = HPF )",
            typeFilter,
            FILTERTYPE_IIRLPF,
            FILTERTYPE_IIRHPF );
```

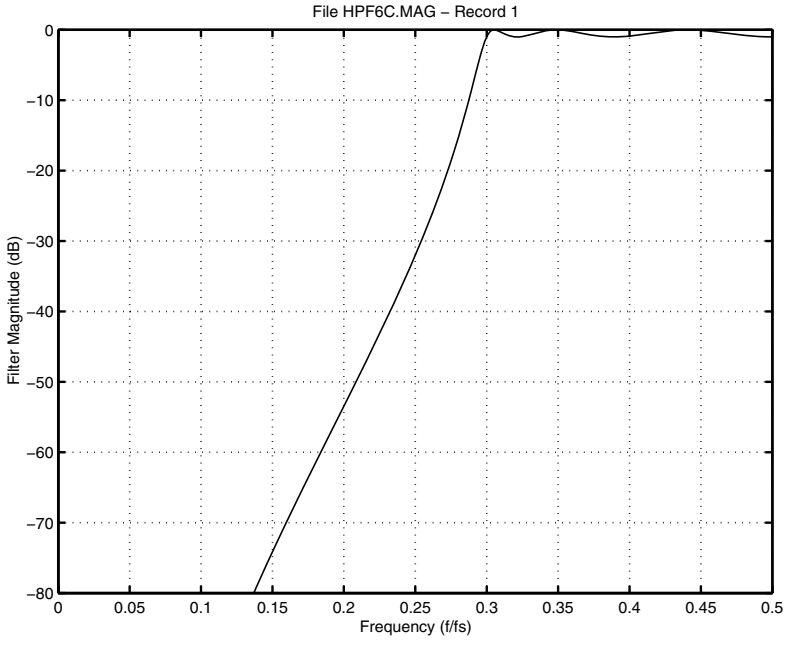
LISTING 4.6 Program IIRFILT uses the **IIRFilter** class and one of the example IIR filters to filter a DSP formatted data record. (*Continued*)

```
        // Create filter with coefficients choosen by user
        String strType;
        Filter<float> *filter = NULL;
        if( typeFilter == FILTERTYPE_IIRLPF )
        {
            strType = “"\nFiltered with lowpass 5th order IIR filter";
            filter = &IIRLPF;
        }
        else
        {
            strType = "\nFiltered with highpass 6th order IIR filter";
            filter = &IIRHPF;
        }

        // Write filtered vector out to file
        do getInput( "Enter filtered output file name", strName );
        while( strName.isEmpty() );
        dspfileOut.openWrite( strName );

        // Filter data one record at a time
        Vector<float> vIn;
        for( int i = 0; i < dspfileIn.getNumRecords(); i++ )
        {
            // Get record
            dspfileIn.read( vIn );
            // Write filtered record
            dspfileOut.write( filter->filter( vIn ) );
            // Reset history between records
            filter->reset();
        }

        // Append to trailer
        strTrailer += strType;
        dspfileOut.setTrailer( strTrailer );

        // Close files
        dspfileIn.close();
        dspfileOut.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 4.6 (*Continued*)

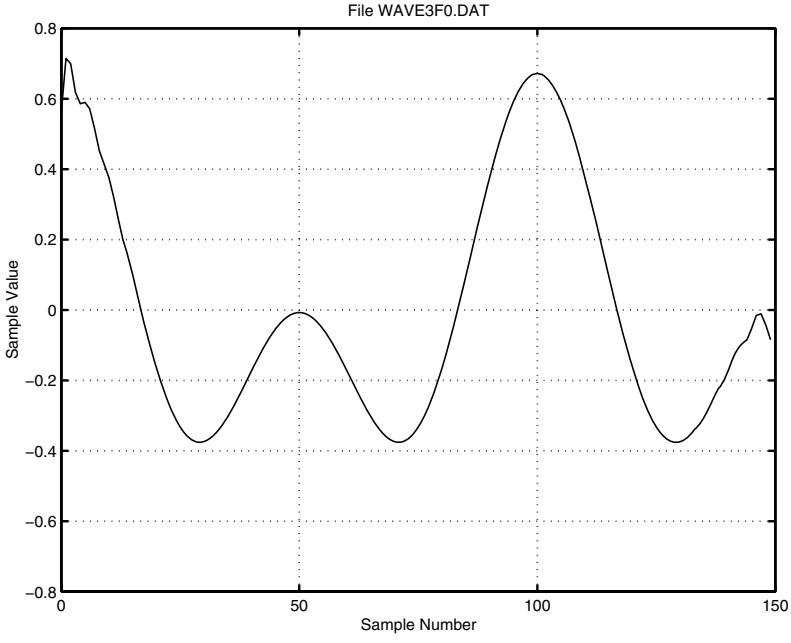


(a)

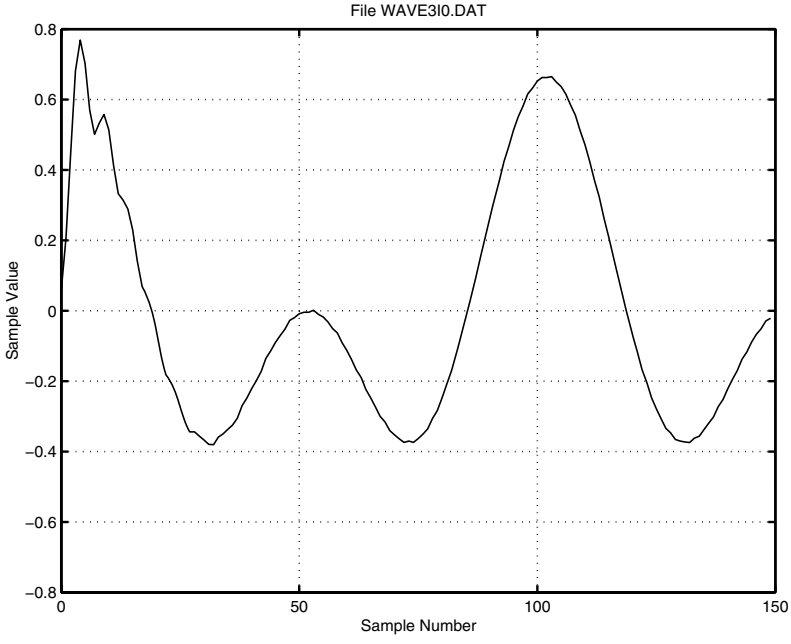


(b)

FIGURE 4.11 Highpass sixth-order Chebyshev IIR filter frequency response. (a) Linear magnitude versus frequency. (b) Log magnitude in decibels versus frequency.

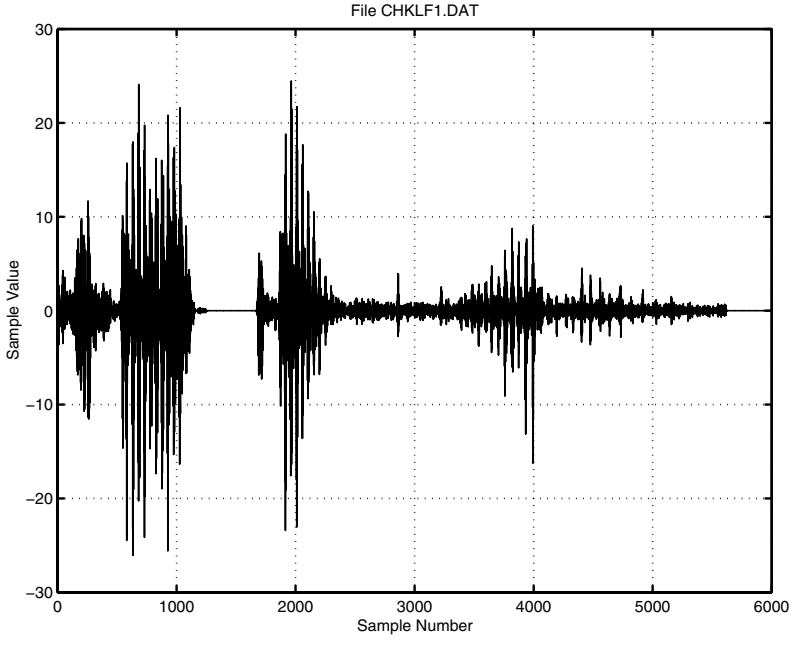


(a)

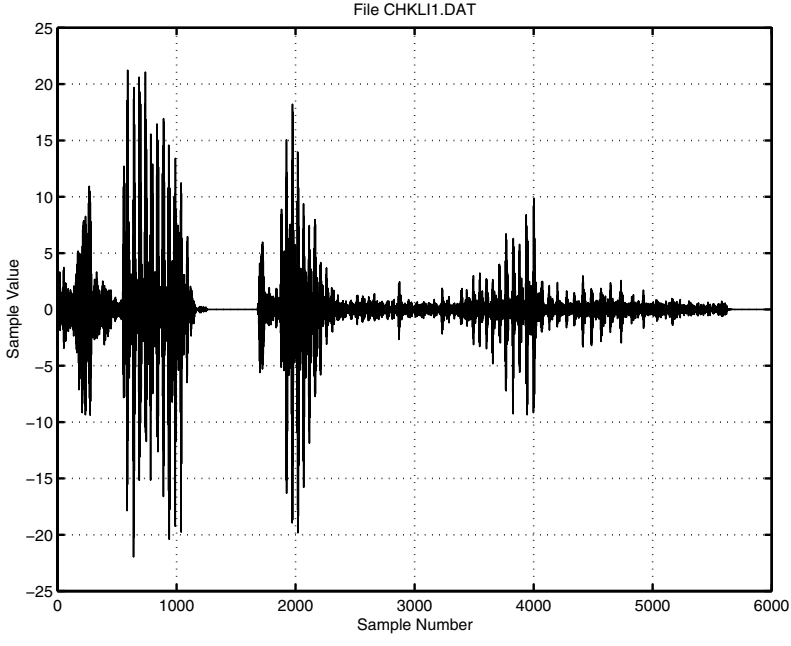


(b)

Figure 4.12 IIR lowpass filtering compared to FIR lowpass filtering. (a) Lowpass filtered WAVE.DAT using 35-point FIR filter [same as Figure 4.5(b)]. (b) IIR lowpass filter using fifth order elliptic filter shown in Figure 4.10.



(a)



(b)

Figure 4.13 IIR highpass filtering compared to FIR highpass filtering. (a) High-filtered CHKL.DAT using 35-point FIR filter [same as Figure 4.6(b)]. (b) Highpass filter using the sixth-order Chebyshev IIR filter shown in Figure 4.11.

```
f/fs = 0.010000
f/fs = 0.020000
f/fs = 0.400000

Enter filter type (0 = LPF, 1 = HPF) [0...1]: 0

Enter output file name: WAVE3I0.DAT
```

Figure 4.12(b) shows the result of the above use of IIRFILT (data file WAVE3I0.DAT). Figure 4.12(a) shows the 35-point FIR lowpass result for comparison, since the two filters have approximately the same magnitude response. Note that both filters remove the highest frequency in the WAVE3.DAT data record. The results are not identical, however. The IIR result has a considerably longer startup transient and is shifted slightly with respect to the FIR result. This is because the FIR result was generated using the vector **FIRFilter** function, which compensates for the delay of the FIR filter (the delay is half the filter length). Because the IIR filter delay depends on the coefficients used and the input frequency, this compensation is not possible for the general-purpose **IIRFilter** function.

It is interesting to compare the computations required for the IIR lowpass filter and the FIR lowpass filter. If the response of the fifth-order IIR filter is acceptable, then the IIR filter can be implemented with only 10 multiply/adds per input sample compared to 35 multiply/adds per input sample for the FIR filter. Thus, the use of an IIR filter instead of an FIR filter can represent a significant cost savings or performance enhancement if the multiply/add time dominates the computation. Unfortunately, on many modern computers multiplies are as fast (if not faster) than other operations, and the time required to update the history array and do the additional data movement associated with the IIR filter can make the time to compute a fifth-order IIR output sample almost equal to the time required to compute a 35-point FIR output sample.

Figure 4.13 shows a similar comparison of the FIR highpass filter and the IIR highpass filter using the CHKL.DAT voice data record. Note that these two results are similar inspite of the fact that the magnitude response of the two filters does not meet the same filter specifications. Careful examination of the two results does reveal some differences in the shape of the two high-frequency envelopes.

4.4 REAL-TIME FILTERS

Real-time filters are filters which are implemented so that a continuous stream of input samples can be filtered to generate a continuous stream of output samples. In many cases, real-time operation restricts the filter to operate on the input samples individually and generate one output sample for each input sample. Multiple memory accesses to the same input data are not possible (as was used in the vector **FIRFilter** function), because only the current input and output are available to the filter at any given instant in time. Thus, some type of history must be stored and updated with each new input sample. The man-

agement of the filter history almost always takes a portion of the processing time, thereby reducing the maximum sampling rate which can be supported by a particular processor.

The function **IIRFilter** (described in Section 4.2.2) is implemented in a form that can be used for real-time filtering. Suppose that the data **input** and **out** exist, which return an input sample and store an output sample at the appropriate time required by the external hardware. The following code can be used with the **IIRFilter** function to perform continuous-time filtering:

```
for(;;)
    out = filter->filterElement( input );
```

In the above infinite loop **for** statement, filter points to a **Filter** class that contains the coefficients, filter length, and history pointer required by **IIRFilter.** The total time required to execute the **in, IIRFilter,** and store **out** must be less than the filter sampling rate in order to ensure that output and input samples are not lost. The next section describes a function that performs floating-point FIR real-time filtering, and Section 4.4.2 gives a simple example of simulated IIR and FIR real-time filtering.

4.4.1 FIR Real-Time Filters

Figure 4.14 shows a block diagram of the FIR real-time filter implemented by the function **FIRFilter.** Real-time filtering is implemented by **FIRFilter** (shown in Listing 4.7) in a manner similar to the **IIRFilter** function discussed in Section 4.2.2 except that multiple sections are not required, since the FIR filter is implemented in direct form. Given the data variables **input** and **out,** a continuous real-time FIR filter could be implemented as follows:

```
for(;;)
    out = filter->filterElement( input );
```

The **FIRFilter** class implements the FIR filter using the history pointer and coefficients in the **Filter** class (pointed to by **filter**). The history array is allocated on the first use of **FIRFilter** and is used to store the previous N - 1 input samples (where N is the number of FIR filter coefficients). The last few lines of code in **filterElemnet** implements the multiplies and accumulates required for the FIR filter of length N (or **m_length** in **filter**). As the history pointer is advanced by using a post increment, the coefficient pointer is post decremented. This effectively time reverses the coefficients so that a true convolution is implemented. On some signal processing microprocessors, post decrement is not implemented efficiently so this code becomes less efficient. Improved efficiency can be obtained in this case by storing the filter coefficients in a time-reversed order. Note that if the coefficients are symmetrical, as for simple linear phase lowpass filters, then the time-reversed order and normal order are identical. After the **for** loop and N - 1 multiplies have been completed, the history array values are shifted one sample toward **history[0]**

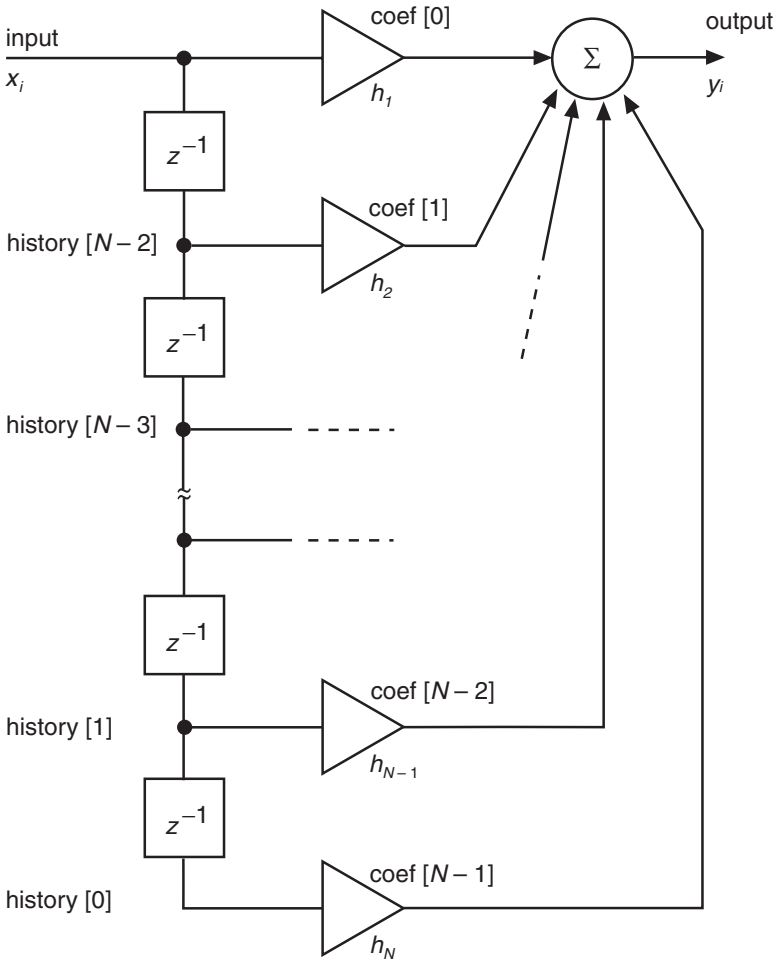


FIGURE 4.14 Block diagram of real-time *N* tap FIR filter structure as implemented by function **filterElement**.

so that the new input sample can be stored in **history[*N* - 1].** The C++ implementation uses pointers extensively for maximum efficiency.

4.4.2 Real-Time Filtering Example

The program REALTIME shown in Listing 4.8 is an example use of the **FIRFilter** and **IIRFilter** real-time filtering functions. The program generates a sinewave chirp signal one sample at a time and filters the chirp samples with one of the four filters defined in FILTER.CPP (filter types 1–4). The chirp without filtering can be generated by selecting

```
// Implementation of abstract base class function (see Filter)
virtual Type filterElement( const Type input )
{
    // Start at beginning of history
    Type *ptrHist = m_hist;
    Type *ptrHist1 = m_hist;

    // point to last coefficient
    Type *ptrCoef = m_coef + m_length - 1;

    // Form output accumulation
    Type output = *ptrHist++ * *ptrCoef—;
    for( int i = 2; i < m_length; i++ )
    {
        // Update history array
        *ptrHist1++ = *ptrHist;
        output += *ptrHist++ * *ptrCoef—;
    }

    // Input tap
    output += input * *ptrCoef;
    // Last history
    *ptrHist1 = input;
    return output;
}
```

LISTING 4.7 Member function **filterElemnet** (part of the **FIRFilter** class in FILTER.H) used to implement the real-time FIR filter shown in Figure 4.14.

filter type 0. The unfiltered chirp signal is shown in Figure 4.15. The instantaneous frequency sweeps from 0 to $f_s/2$ in the 500 samples of the data record. The unusual amplitude variation of the high-frequency portion of the chirp is due to the linear interpolation performed when the signal was plotted. The following computer dialogue illustrates the REALTIME program when the FIR lowpass filter is selected:

```
Enter number of samples to generate [2...10000] : 500

Filters available:

0 - No Filter (chirp signal input)
1 - FIR 35 point lowpass, Pass 0-0.2, Stop 0.25-0.5
2 - FIR 35 point highpass, Pass 0.3-0.5, Stop 0-0.25
3 - IIR Elliptic lowpass, Pass 0-0.2, Stop 0.25-0.5
4 - IIR Chebyshev highpass, Pass 0.3-0.5, Stop 0-0.25
Enter filter type [0...4] : 1
Enter output file name : RT1.DAT
```

```
//////////////////////////////////////////////////////////////
//
// realtime.cpp - Demonstrate real-time filtering
//
//////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "filter.h"
#include "fircoefs.h"
#include "iircoefs.h"

//////////////////////////////////////////////////////////////
//
// int main()
//   Demonstrates real-time filtering using FIRFilter and
//   IIRFilter. A chirp signal is generated one sample at a
//   time before each filterElement call.
//
//  Returns:
//   0 -- Success
//
//////////////////////////////////////////////////////////////
int main()
{
    try
    {
       DSPFile dspfileOut;
       String strName;
       String strTrailer;

       // Create sample vector
       int length = 0;
       getInput("Enter number of samples to generate",length, 2,65535);
       Vector<float> vSignal( length );

       // Get type of filter to use of several defined in filter.h
       int typeFilter = 0;
       cout<<"Filters available:\n";
       cout<<" 0-No Filter (chirp signal input)\n";
       cout<<" 1-FIR 35 point lowpass, Pass 0-0.2, Stop 0.25-0.5\n";
       cout<<" 2-FIR 35 point highpass, Pass 0.3-0.5, Stop 0-0.25\n";
       cout<<" 3-IIR Elliptic lowpass, Pass 0-0.2, Stop 0.25-0.5\n";
       cout<<" 4-IIR Chebyshev highpass, Pass 0.3-0.5, Stop 0-0.25\n";
```

LISTING 4.8 Program REALTIME uses the **IIRFilter** or **FIRFilter** class to demonstrate real-time filtering of a chirp signal. (*Continued*)

```
getInput( “Enter type of filter”, typeFilter, 0, 4 );

Filter<float> *filter = NULL;
if( typeFilter == 0 )
    strTrailer = “Chirp Signal without filter”;
else if( typeFilter == 1 )
{
    filter = &FIRLPF;
    strTrailer = “Chirp after FIR 35 point lowpass filter”;
}
else if( typeFilter == 2 )
{
    filter = &FIRHPF;
    strTrailer = “Chirp after FIR 35 point highpass filter”;
}
else if( typeFilter == 3 )
{
    filter = &IIRLPF;
    strTrailer = "Chirp after IIR Elliptic lowpass filter";
}
else if( typeFilter == 4 )
{
    filter = &IIRHPF;
    strTrailer = "Chirp after IIR Chebyshev highpass";
}

// Generate and filter the data

// PI / 2 * length
double arg = 2.0 * atan( 1.0 ) / (double)length;

for( int i = 0; i < length ; i++ )
{
    float input = (float)( sin( (double)i*(double)i*arg));
    if( filter == NULL )
        vSignal[i] = input;
    else
        vSignal[i] = filter->filterElement( input );
}

// Write filtered vector out to file
do getInput( "Enter filtered output file name", strName );
while( strName.isEmpty() );
dspfileOut.openWrite( strName );
dspfileOut.write( vSignal );
```

LISTING 4.8 (*Continued*)

```
        dspfileOut.setTrailer( strTrailer );
        dspfileOut.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 4.8 (*Continued*)

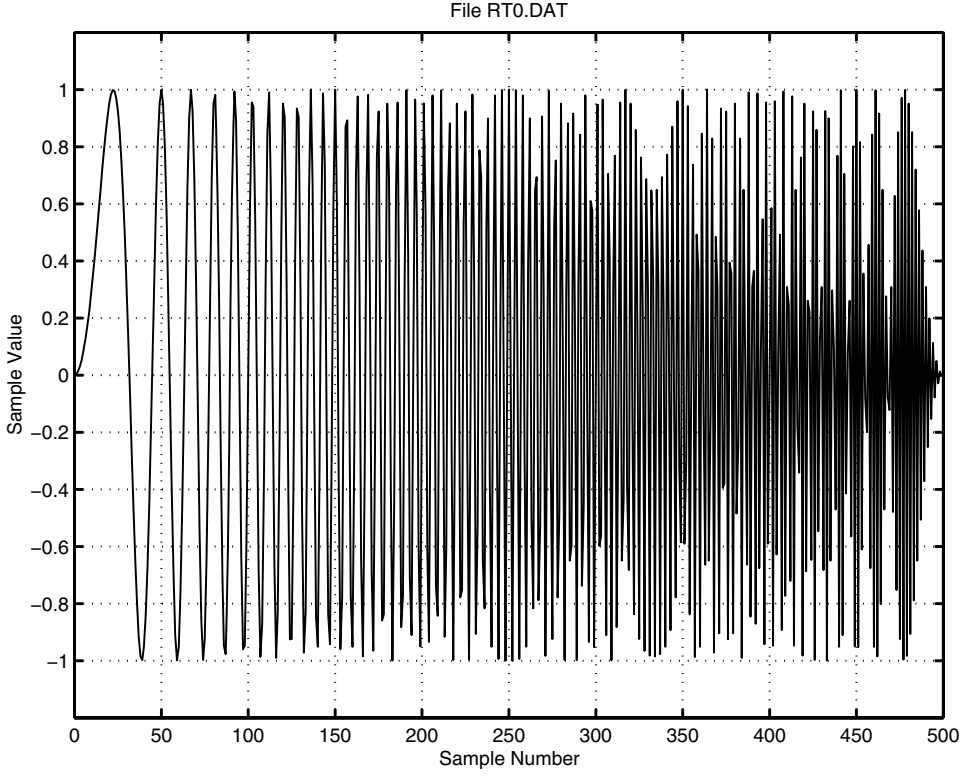
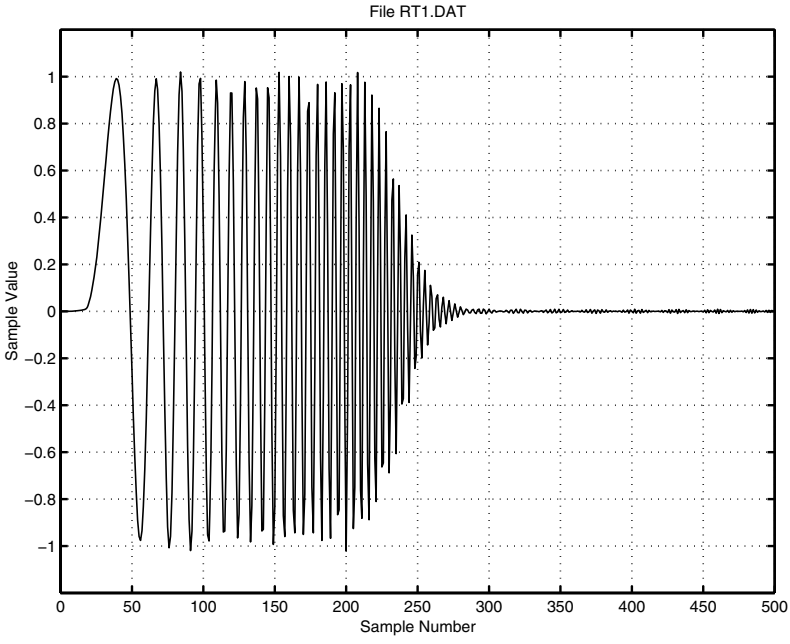
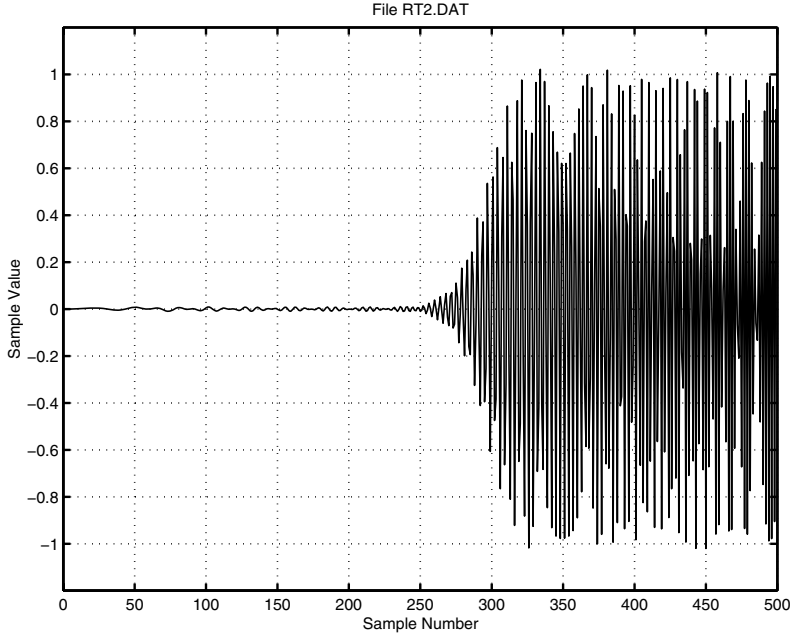


FIGURE 4.15 Chirp signal used to demonstrate real-time filtering generated by program REALTIME (type = 0).

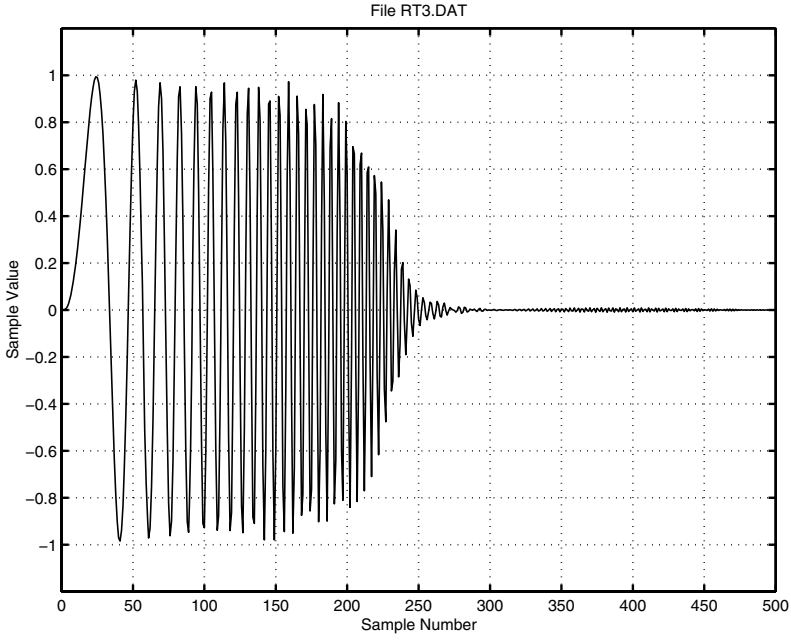


(a)

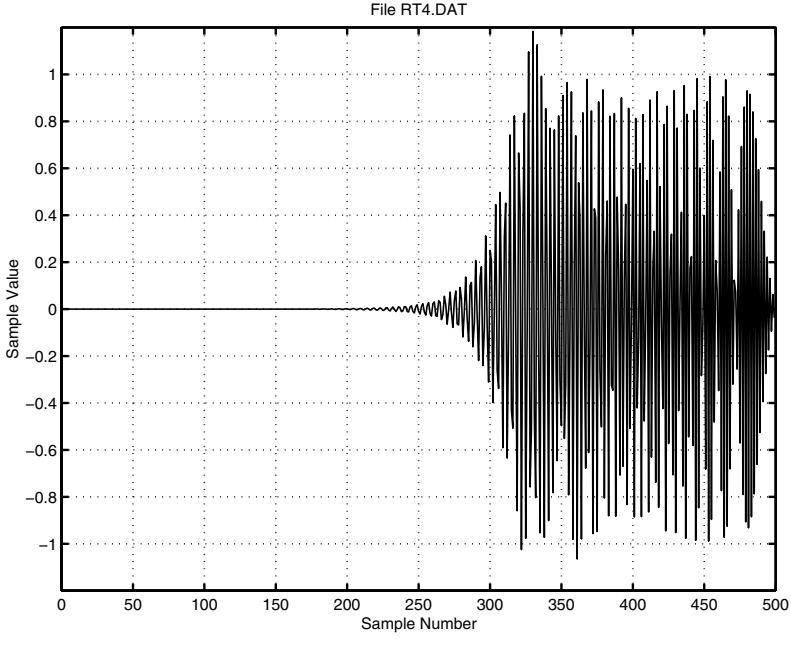


(b)

FIGURE 4.16 FIR real-time filtering of chirp signal using program REAL-TIME. (a) FIR 35-point lowpass filter (type = 1). (b) FIR 35-point highpass filter (type = 2).



(a)



(b)

FIGURE 4.17 IIR real-time filtering of chirp signal using program REAL-TIME. (a) Fifth-order elliptic lowpass (type = 3). (b) Sixth-order Chebyshev highpass (type = 4).

The output file generated by the above dialogue (file RT1.DAT) is shown in Figure 4.16(a). Figure 4.16(b) shows the result of the highpass FIR filter, and Figure 4.17 shows the two IIR filter results. The approximate magnitude response and the startup delay of each filter is evident.

4.5 INTERPOLATION AND DECIMATION

Many signal processing applications require that the output sampling rate be different from the input sampling rate. Sometimes one section of a system can be made more efficient if the sampling rate is lower (such as when simple FIR filters are involved or in data transmission). In other cases, the sampling rate must be increased so that the spectral details of the signal can be easily identified. In either case, the input sampled signal must be resampled to generate a new output sequence with the same spectral characteristics but at a different sampling rate. Increasing the sampling rate is called *interpolation* or *upsampling.* Reducing the sampling rate is called *decimation* or *downsampling.* Normally, the sampling rate of a bandlimited signal can be interpolated or decimated by integer ratios such that the spectral content of the signal is unchanged. By cascading interpolation and decimation, the sampling rate of a signal can be changed by any rational fraction, P/M, where P is the integer interpolation ratio and M is the integer decimation ratio. Interpolation and decimation can be performed using filtering techniques (as described in this section) or by using the fast Fourier transform (see Chapter 5 or the texts by Brigham, or Crochiere and Rabiner).

Decimation is perhaps the simplest resampling technique because it involves reducing the number of samples per second required to represent a signal. If the input signal is strictly bandlimited such that the signal spectrum is zero for all frequencies above $f_s/(2M)$, then decimation can be performed by simply retaining every Mth sample and discarding the M - 1 samples in between. In fact, the following short program (which can be found on the disk as file DECIM.CPP) can be used to decimate a DSP data record:

```
////////////////////////////////////////////////////////////////////////
//
// decim.cpp - Decimate samples in DSPFile
//
////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"

////////////////////////////////////////////////////////////////////////
//
// int main()
// Decimates samples in DSPFile by only writing samples
// at the decimation frequency.
```

```
//
// Returns:
// 0 -- Success
//
////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Open the input file
        do getInput( "Enter file to decimate", strName );
        while( strName.isEmpty() || dspfile.isFound(strName) == false);
        dspfile.openRead( strName );

        // Read in file
        Vector<float> vIn;
        dspfile.read( vIn );
        dspfile.close();

        // Get decimation ration
        int dec = 0;
        getInput("Enter decimation ratio", dec, 2, vIn.length() - 1 );
        int lenOut = vIn.length() / dec;

        // Decimate the data in place
        for( int i = 0; i < lenOut; i++ )
            vIn[i] = vIn[i * dec];

        // Write one record of floats
        do getInput( "Enter output file name", strName );
        while( strName.isEmpty() );
        dspfile.openWrite( strName );
        dspfile.write( vIn( 0, lenOut ) );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

Unfortunately, the spectral content of a signal above $f_s/(2M)$ is rarely zero and the aliasing caused by the simple decimation illustrated above almost always causes trouble. Even when the desired signal is zero above $f_s/(2M)$, some amount of noise is usually present and will alias into the low-frequency signal spectrum. Aliasing due to decimation can be avoided by lowpass filtering the signal before the samples are decimated. For example, when $M = 2$, the 35-point lowpass FIR filter introduced in Section 4.2 can be used to eliminate almost all spectral content above $0.25f_s$ (the attenuation above $0.25\ f_s$ is greater than 40 dB). The above decimation program could then be used to reduce the sampling rate by a factor of two. An IIR lowpass filter (discussed in Section 4.3) could also be used to eliminate the frequencies above $f_s/(2M)$ as long as linear phase response is not required.

Interpolation is the process of computing new samples in the intervals between existing data points. Classical interpolation (which was used before calculators and computers) involves estimating the value of a function between existing data points by fitting the data to a low-order polynomial. For example, *linear* (first order) or *quadratic* (second order) polynomial interpolation is often used. The primary attraction of polynomial interpolation is computational simplicity. The primary disadvantage is that in signal processing, the input signal must be restricted to a very narrow band so that the output will not have a large amount of aliasing. Thus, bandlimited interpolation using digital filters is usually the method of choice in digital signal processing. Bandlimited interpolation by a factor P:1 (see Figure 4.18 for an illustration of 3:1 interpolation) involves the following conceptual steps:

1. Make an output sequence P times longer than the input sequence. Place the input sequence in the output sequence every P samples and place P - 1 zero values between each input sample. This is called *zero-packing* (as opposed to *zero-padding*). The zero values are located where the new interpolated values will appear. The effect of zero-packing on the input signal spectrum is to replicate the spectrum P times within the output spectrum. This is illustrated in Figure 4.18(a) where the output sampling rate is three times the input sampling rate.
2. Design a lowpass filter capable of attenuating the undesired P - 1 spectrums above the original input spectrum. Ideally, the passband should be from 0 to $f_s'/(2P)$ and the stopband should be from $f_s'/(2P)$ to $f_s'/2$ (where f_s' is the filter sampling rate which is P times the input sampling rate). A more practical interpolation filter has a transition band that is centered about $f_s'/(2P)$. This is illustrated in Figure 4.18(b). The passband gain of this filter must be equal to P to compensate for the inserted zeroes so that the original signal amplitude is preserved.
3. Filter the zero-packed input sequence using the interpolation filter to generate the final P:1 interpolated signal. Figure 4.18(c) shows the resulting 3:1 interpolated spectrum. Note that the two repeated spectrums are attenuated by the stopband attenuation of the interpolation filter. In general, the stopband attenuation of the filter

must be greater than the signal-to-noise ratio of the input signal in order for the interpolated signal to be a valid representation of the input.

Figure 4.18(d) also illustrates 2:1 decimation after the 3:1 interpolation. Figure 4.18(d) shows the spectrum of the final signal, which has a sampling rate that is 1.5 times the input sampling rate. Because no lowpass filtering (other than the filtering by the 3:1 interpolation filter) is performed before the decimation shown, the output signal near f_s"/2 has an unusually shaped power spectrum due to the aliasing of the 3:1 interpolated spectrum. If this aliasing causes a problem in the system that processes the interpolated output signal, it can be eliminated by either lowpass filtering the signal before decimation or by designing the interpolation filter to further attenuate the replicated spectra.

The interpolation filter used to create the interpolated values can be an IIR or FIR lowpass filter. However, if an IIR filter is used, the input samples are not preserved exactly because of the nonlinear phase response of the IIR filter. FIR interpolation filters can be designed such that the input samples are preserved, which also results in some computational savings in the implementation. For this reason, only the implementation of FIR interpolation will be considered further.

4.5.1 FIR Interpolation

The FIR lowpass filter required for interpolation can be designed using the McClellan-Parks program (see Section 4.2) or by using the simpler windowing techniques. In this section, a Kaiser window is used to design 2:1 and 3:1 interpolators. The FIR filter length must be odd so that the filter delay is an integer number of samples so that the input samples can be preserved. The passband and stopband must be specified such that the center coefficient of the filter is unity (the filter gain will be P) and every P coefficients on each side of the filter center are zero. This ensures that the original input samples are preserved because the result of all the multiplies in the convolution is zero except for the center filter coefficient, which gives the input sample. The other P - 1 output samples between each original input sample are created by convolutions with the other coefficients of the filter. The following passband and stopband specifications will be used to illustrate a P:1 interpolation filter:

Passband frequencies:	0–0.8 $f_s/(2/P)$
Stopband frequencies:	1.2 $f_s/(2P)$ - 0.5 f_s
Passband gain:	P
Passband ripple:	< 0.03 dB
Stopband attenuation:	> 56 dB

The filter length was determined to be 16P - 1 using Equation 4.2 (rounding to the nearest odd length) and the passband and stopband specifications. Greater stopband attenuation or a smaller transition band can be obtained with a longer filter. The interpolation filter coefficients are obtained by multiplying the Kaiser window coefficients by the ideal

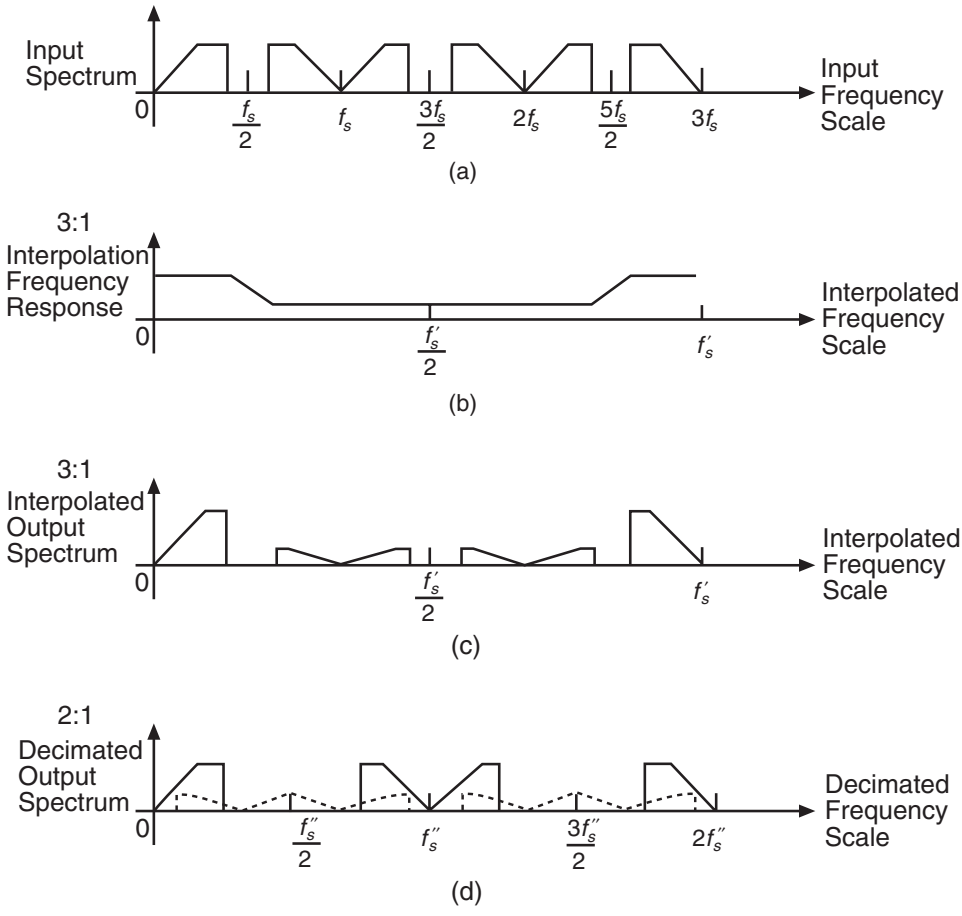


FIGURE 4.18 Illustration of 3:1 interpolation followed by 2:1 decimation. The aliased input spectrum in the decimated output is shown with a dashed line. (a) Example real input spectrum. (b) 3:1 interpolation filter response ($f_s' = 3f_s$). (c) 3:1 interpolated spectrum. (d) 2:1 decimated output ($f_s'' = f_s'/2$).

lowpass filter coefficients. The ideal lowpass coefficients for a very long odd-length filter with a cutoff frequency of $f_s/2P$ are given by the following sinc function:

$$c_k = \frac{P \sin(k\pi/P)}{k\pi} \tag{4.14}$$

Note that the original input samples are preserved, since the coefficients are zero for all $k = nP$, where n is an integer greater than zero and $c_0 = 1$. Very poor stopband attenuation would result if the above coefficients were truncated by using the $16P - 1$ coefficients

where $|k| < 8P$ (effectively multiplying the sinc function by a rectangular window, which would have a stopband attenuation of about 13 dB). However, by multiplying these coefficients by the appropriate Kaiser window, the stopband and passband specifications can be realized. The symmetrical Kaiser window, w_k, is given by the following expression:

$$w_k = \frac{I_0\left\{\beta\sqrt{1-\left(1-\frac{2k}{N-1}\right)^2}\right\}}{I_0(\beta)} \tag{4.15}$$

Where $I_0(ß)$ is a modified zero order Bessel function of the first kind and ß is the Kaiser window parameter, which determines the stopband attenuation. The empirical formula for ß when A_{stop} is greater than 50 dB is ß = 0.1102*(A_{stop} − 8.71). Thus, for a stopband attenuation of 56 dB, ß = 5.21136. Figure 4.19(a) shows the frequency response of the resulting 31-point 2:1 interpolation filter, and Figure 4.19(b) shows the frequency response of the 47-point 3:1 interpolation filter.

Listing 4.9 shows the program INTERP, which implements 2:1 or 3:1 interpolation on the first record in a DSP data file using the filters shown in Figure 4.19. The 47 filter coefficients of the 3:1 interpolation filter are stored in the static array **interp3,** and the 31 filter coefficients of the 2:1 interpolation filter are stored in the static array **interp2.** Both of these arrays are initialized in the beginning of the program as shown in Listing 4.9. In order to implement the interpolation filters efficiently, each set of filter coefficients are decimated by a factor of 2 or 3 (the interpolation ratio) as follows:

```
if(ratio == 2) {
    float coef2[FILTER_LENGTH];
    for( int i = 0; i < FILTER_LENGTH; i++ )
        coef2[i] = INTERP2[2*i];
    FIRFilter<float> interp2( coef2, FILTER_LENGTH );
}
else {    // ratio = 3 case
    float coef31[FILTER_LENGTH];
    for( int i = 0; i < FILTER_LENGTH; i++ )
        coef31[i] = INTERP3[3*i];
    FIRFilter<float> interp31( coef31, FILTER_LENGTH );

    float coef32[FILTER_LENGTH];
    for( i = 0; i < FILTER_LENGTH; i++ )
        coef32[i] = INTERP3[3*i+1];
    FIRFilter<float> interp32( coef32, FILTER_LENGTH );
}
```

In each case, a **Filter** class is initialized to have the decimated coefficients. If 2:1 interpolation is selected, only one set of 16 coefficients is required (the odd coefficients in **interp2**). If 3:1 interpolation is selected, two **Filter** structures are initialized each with 16 coefficients. Each of the **Filter** classes are then used individually with the **filterElemnet**

```
////////////////////////////////////////////////////////////////////////////
//
// interp.cpp - Interpolation using filters
//
////////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "filter.h"
////////////////////////////////////////////////////////////////////////////
//
// 2:1 interpolation coefficients, PB 0-0.2, SB 0.3-0.5
//
////////////////////////////////////////////////////////////////////////////
static float INTERP2[] =
{
    -0.00258821F, 0.0F,  0.00748852F, 0.0F, -0.01651064F, 0.0F,
     0.03176119F, 0.0F, -0.05692563F, 0.0F,  0.10079526F, 0.0F,
    -0.19532167F, 0.0F,  0.63082207F, 1.0F,  0.63082207F, 0.0F,
    -0.19532167F, 0.0F,  0.10079526F, 0.0F, -0.05692563F, 0.0F,
     0.03176119F, 0.0F, -0.01651064F, 0.0F,  0.00748852F, 0.0F,
    -0.00258821F
};
////////////////////////////////////////////////////////////////////////////
//
// 3:1 interpolation coefficients, PB 0-0.133, SB 0.2-0.5
//
////////////////////////////////////////////////////////////////////////////
static float INTERP3[] =
{
    -0.00178662F, -0.00275941F, 0.0F,  0.00556927F,  0.00749929F, 0.0F,
    -0.01268113F, -0.01606336F, 0.0F,  0.02482278F,  0.03041984F, 0.0F,
    -0.04484686F, -0.05417098F, 0.0F,  0.07917613F,  0.09644332F, 0.0F,
    -0.14927754F, -0.19365910F, 0.0F,  0.40682136F,  0.82363913F, 1.0F,
     0.82363913F,  0.40682136F, 0.0F, -0.19365910F, -0.14927754F, 0.0F,
     0.09644332F,  0.07917613F, 0.0F, -0.05417098F, -0.04484686F, 0.0F,
     0.03041984F,  0.02482278F, 0.0F, -0.01606336F, -0.01268113F, 0.0F,
     0.00749928F,  0.00556927F, 0.0F, -0.00275941F, -0.00178662F
};
////////////////////////////////////////////////////////////////////////////
//
// int main()
//   Demonstrates 2:1 and 3:1 FIR filter interpolation
//   using two interpolation filters and multiple calls
//   to the real time filter function filterElement().
```

LISTING 4.9 Program INTERP used to interpolate a DSP data record by a factor of 2 or 3 using the filters shown in Figure 4.18. (*Continued*)

```
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
     DSPFile dspfileIn;
     DSPFile dspfileOut;
     String strName;
     String strTrailer;

     // Open the input file
     do getInput( "Enter file to filter", strName );
     while( strName.isEmpty() || dspfileIn.isFound( strName ) == false );
     dspfileIn.openRead( strName );
     dspfileIn.getTrailer( strTrailer );
     int lenIn = dspfileIn.getRecLen();

     // Print trailer
     if( strTrailer.isEmpty() == false )
         cout << "File trailer:\n" << strTrailer << endl;

     // Get ratio of interpolation
     int ratioInterpolation = 0;
     getInput(
         "Enter ratio of interpolation",
         ratioInterpolation,
         2,
         3 );

     // Write interpolated vector out to file
     do getInput( "Enter interpolated output file name", strName );
     while( strName.isEmpty() );
     dspfileOut.openWrite( strName );

     // Filter with interpolation ratio
     String strRatio;
     Vector<float> vIn;
     const int FILTER_LENGTH = 16;
     if( ratioInterpolation == 2 )
     {
         strRatio = "FIR Interpolated 2:1\n";
```

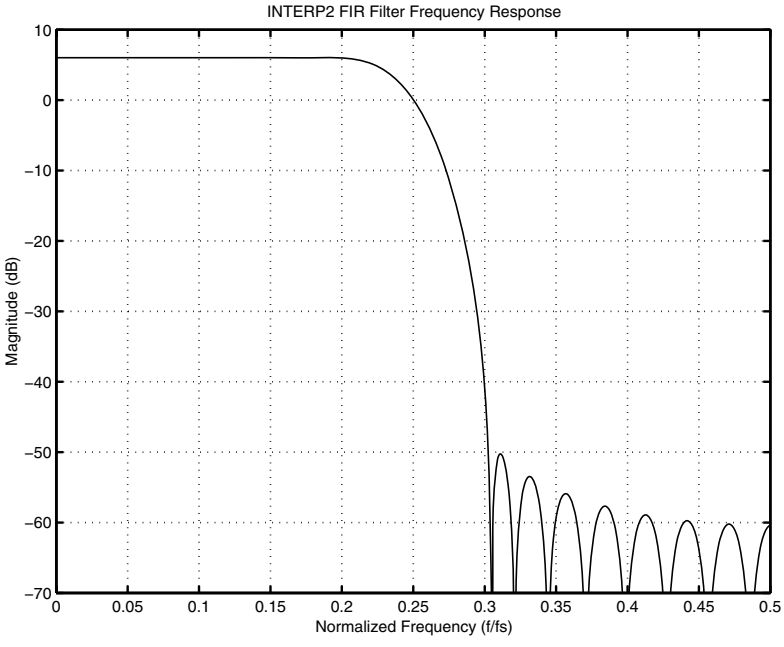
LISTING 4.9 (*Continued*)

```
 // Decimate coefficients of filter
 float coef2[FILTER_LENGTH];
 for( int i = 0; i < FILTER_LENGTH; i++ )
     coef2[i] = INTERP2[2*i];
 FIRFilter<float> interp2( coef2, FILTER_LENGTH );

 // Create output vector
 Vector<float> vOut( ratioInterpolation * lenIn );

 // Read in file record by record
 for( int j = 0; j < dspfileIn.getNumRecords(); j++ )
 {
     // Read in vector to interpolate
     dspfileIn.read( vIn );

     // Set elements of output vector to zero
     vOut = 0.0f;

     // Interpolate the data by calls to fir_filter
     for( i = 0; i < lenIn; i++ )
     {
         if( i >= 8 )
             vOut[2*i] = vIn[i-8];
         vOut[2*i+1] = interp2.filterElement( vIn[i] );
     }
     dspfileOut.write( vOut );
 }
}
else
{
 strRatio = "FIR Interpolated 3:1\n";

 // Decimate coefficients of filter
 float coef31[FILTER_LENGTH];
 for( int i = 0; i < FILTER_LENGTH; i++ )
     coef31[i] = INTERP3[3*i];
 FIRFilter<float> interp31( coef31, FILTER_LENGTH );

 float coef32[FILTER_LENGTH];
 for( i = 0; i < FILTER_LENGTH; i++ )
     coef32[i] = INTERP3[3*i+1];
 FIRFilter<float> interp32( coef32, FILTER_LENGTH );

 // Create output vector
 Vector<float> vOut( ratioInterpolation * lenIn );
```

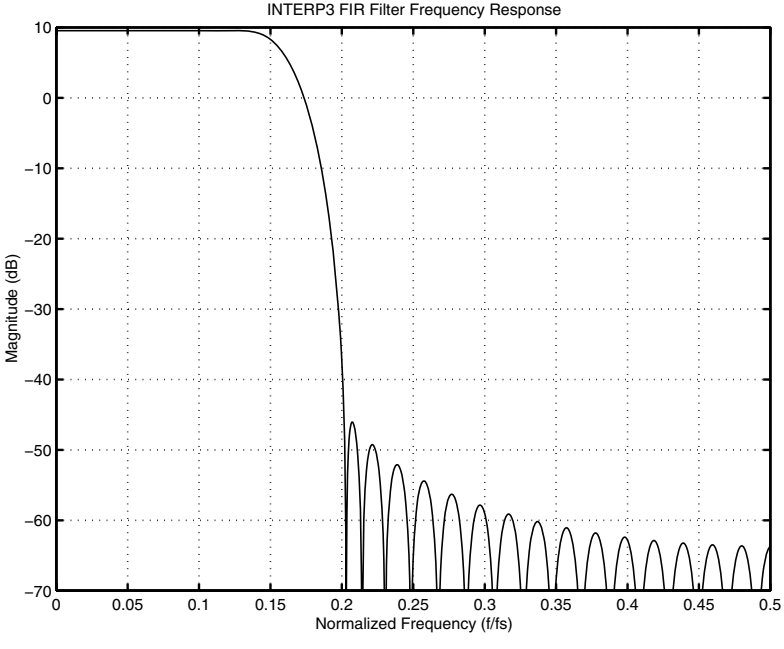
LISTING 4.9 (*Continued*)

```
        // Read in file record by record
        for( int j = 0; j < dspfileIn.getNumRecords(); j++ )
        {
            // Read in vector to interpolate
            dspfileIn.read( vIn );

            // Set elements of output vector to zero
            vOut = 0.0f;

            // Interpolate the data by calls to fir_filter
            for( i = 0; i < lenIn; i++ )
            {
                if( i >= 8 )
                    vOut[3*i] = vIn[i-8];
                vOut[3*i+1] = interp31.filterElement( vIn[i] );
                vOut[3*i+2] = interp32.filterElement( vIn[i] );
            }
            dspfileOut.write( vOut );
        }
       }

       // Append to trailer
       strTrailer += strRatio;
       dspfileOut.setTrailer( strTrailer );
       cout << strTrailer << endl;

       // Close files
       dspfileIn.close();
       dspfileOut.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 4.9 (*Continued*)



(a)



(b)

FIGURE 4.19 (a) Frequency response of 31-point FIR 2:1 interpolation filter (gain = 2 or 6 dB). (b) Frequency response of 47-point FIR 3:1 interpolation filter (gain = 3 or 9.54 dB).

function to create the interpolated values in the **vOut** array. The original input signal is copied without filtering to the **vOut** array every P samples (where P is 2 or 3). Thus, compared to direct filtering using the 31- or 47-point original filters, 15 multiplies for each input sample are saved when interpolation is performed using INTERP. Also, because the **filterElement** function is used to perform the filtering, the same program structure could be used to perform interpolation in real-time with real-time hardware.

Figure 4.20 shows the result of running the INTERP program on the WAVE3.DAT data file (introduced in Section 4.2.1). Figure 4.20 shows the original data. The result of the 3:1 interpolation ratio is shown in Figure 4.20b. Note that the definition of the highest frequency in the original data set (0.4 f_s) is much improved because in Figure 4.20(b) there are 7.5 samples per cycle of the highest frequency. The startup effects and the 23 sample delay of the 47-point interpolation filter are also easy to see in Figure 4.20(b) when compared to Figure 4.20(a). The following computer dialogue shows how the interpolated data (file WAVE33.DAT) was created:

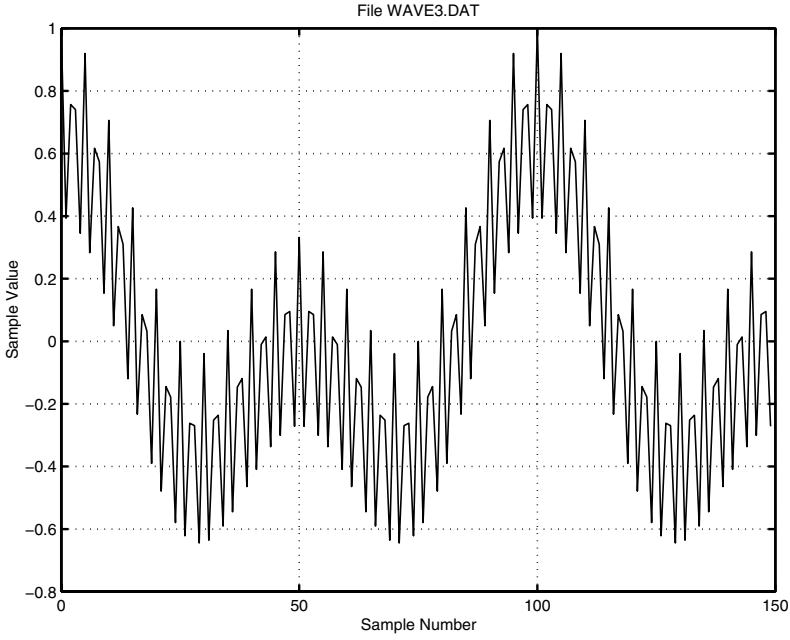
```
Enter input file name : WAVE3.DAT

File trailer:

Signal of length 150 equal to the sum of 3
cosine waves at the following frequencies:
f/fs = 0.010000
f/fs = 0.020000
f/fs = 0.400000

Enter interpolation ratio [2...3] : 3

Enter output file name : WAVE33.DAT
```

4.5.2 FIR Sample Rate Modification and Pitch Shifting

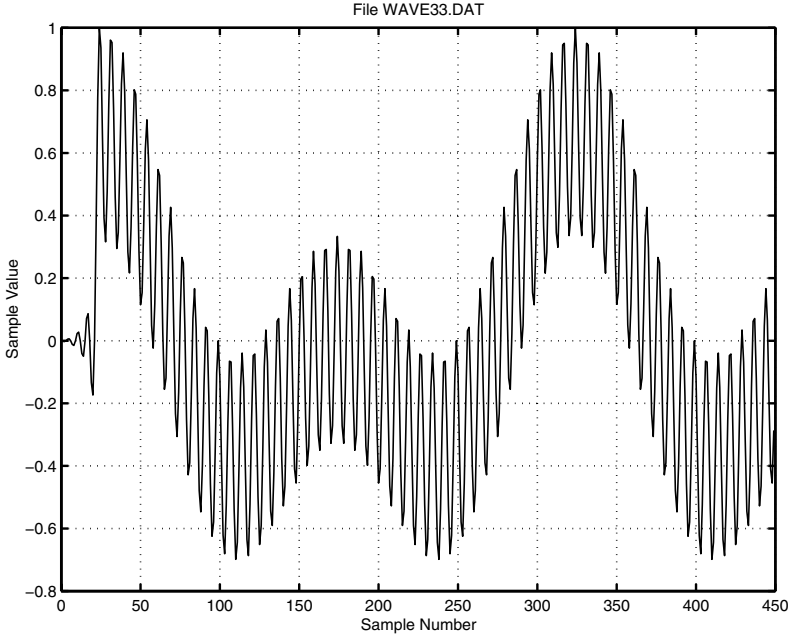
By changing the sampling rate of an audio signal by FIR interpolation (or in combination with decimation) and then playing the digital samples at the original sampling rate, the pitch of the recorded sound will shift. Changing the pitch of a recorded sound is often desired in order to allow it to mix with a new song or for special effects where the original sound is shifted in frequency to a point where the original sound is no longer identifiable. New sounds are often created by a series of pitch shifts and mixing processes.

Different sample rates are used in the audio recording and production process. For example, a 48-KHz sample rate is often used for high-quality recording and mixing. The final result is then converted to a 44.1-KHz sample rate for mass production of audio compact discs. This sample rate conversion requires an interpolation factor of 147 followed by a decimation factor of 160. The example program presented in this section (PSHIFT.CPP) illustrates pitch shifting for audio samples and can also be used for a fixed integer interpolation and decimation factor for sample rate modification.

Pitch shifting is accomplished by interpolating a signal to a new sampling rate and then playing the new samples back at the original sampling rate. If the pitch is shifted



(a)



(b)

FIGURE 4.20 Example of INTERP for 3:1 interpolation. (a) Original WAVE3.DAT [same as Figure 4.5(a)]. (b) 3:1 interpolated WAVE3.DAT.

down (by an interpolation factor greater than one) the new sound will have a longer duration. If the pitch is shifted upward (by an interpolation factor less than one where some decimation is occurring), the sound becomes shorter. Listing 4.10 shows the program PSHIFT.CPP, which can be used to pitch shift a sound up or down by a number of semitones (12 semitones is an octave or a factor of 2 in frequency). It uses a long Kaiser window filter for interpolation of the samples, as was illustrated in the last section. The filter coefficients are calculated in the first part of the PSHIFT program before input and output begins. The filtering is done with two FIR filter functions that are shown in Listing 4.11. The history array is updated only when the interpolation point moves to the next input sample. This requires that the history update be removed from the **filter** function discussed previously. The history is updated by the function **FIRHistoryUpdate**. The coefficients are decimated into short polyphase filters. In the pitch shift case, an interpolation ratio of 500 is performed, and the decimation ratio is determined by the amount of pitch shift selected by the integer variable **key**. A computer dialog for a pitch shift of 5 semitones is as follows:

Enter input file : **chkl.dat**
Enter pitch/sample rate shifted output file name : **chkl5.dat**
Do you want to enter a Musical Key Shift Value (y or n) ? : **y**
Key Shift Value in Semitones (12 = 1 octave) [-60..60] : **5**
Decimation ratio (dec) = 667.42
Interpolation ratio (RATIO) = 500
Poly phase filter size = 24

Note that the decimation ratio is greater than the interpolation ratio becaue the siganl is pitch shift up which makes the final data set shorter than the original. A computer dialog for a sample rate change of the CHKL.DAT data set to 44100 Hz from its original 8000 Hz is as follows:

Enter input file : **chkl.dat**
Enter pitch/sample rate shifted output file name : **chkl441.dat**
Do you want to enter a Musical Key Shift Value (y or n) ? : **n**
Input sample rate [0.01..1e+008] : **8000**
Output sample rate [0.01..1e+008] : **44100**
Max common ratio found (maxTryFound) = 100
Decimation ratio (dec) = 80
Interpolation ratio (RATIO) = 441
Polyphase filter size = 24

```
////////////////////////////////////////////////////////////////////////////////
//
// pshift.cpp - Kaiser Window Pitch Shift/Sample Rate Change Algorithm
//
////////////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "filter.h"

////////////////////////////////////////////////////////////////////////////////
//
// Constants
//
////////////////////////////////////////////////////////////////////////////////

// Passband specified, larger makes longer filters
const float PERCENT_PASS = 85.0f;

// Minimum attenuation in stopbands (dB), larger make long filters
const float ATT = 60.0f;

////////////////////////////////////////////////////////////////////////////////
//
// int main()
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
      DSPFile dspfileIn;
      DSPFile dspfileOut;
      String strName;
      float beta;

      // Open the input file
      do getInput( "Enter input file", strName );
      while( strName.isEmpty() || dspfileIn.isFound( strName ) == false );
      dspfileIn.openRead( strName );
```

LISTING 4.10 Program PSHIFT.CPP, which performs pitch shifting on audio samples or general sample rate conversion. (*Continued*)

```
// Open the output file
do getInput("Enter pitch/sample rate shifted output file name", strName );
while( strName.isEmpty() );
dspfileOut.openWrite( strName );

// Read in a user key value or input/output sample rates
String strKey;
getInput("Do you want to enter a Musical Key Shift Value (y or n) ?", strKey );
if(strKey.isEmpty()) strKey = "n";

// Set interpolation ratio default
int RATIO = 500;

float dec = 1.0f;
if(strKey[0] == 'y')
{
    int key = 0;
    getInput("Key Shift Value in Semitones (12 = 1 octave)", key, -60, 60 );

// Decimation ratio for key semitones shift
    dec = (float)RATIO* pow( 2.0, 0.0833333333 * key );
}
else
{
    float inFreq = 1.0f;
    getInput("Input sample rate", inFreq, 0.01f, 1.e8f );
    float outFreq = 1.0f;
    getInput("Output sample rate", outFreq, 0.01f, 1.e8f);

    // try all RATIOs to 1000 until a common factor
    int maxTryFound = 1;
    for(int tryRatio = 1 ; tryRatio <= 1000 ; tryRatio++)
    {
        float frin = inFreq / tryRatio;
        int irin = (int) frin;
        float ein = frin - (float)irin;
        float frout = outFreq / tryRatio;
        int irout = (int) frout;
        float eout = frout - (float)irout;
        if(ein < 1.e-8 && eout < 1.e-8) {
            maxTryFound = tryRatio;
        }
    }
    cout << "Max common ratio found (maxTryFound) = " << maxTryFound << endl;
    float maxIOSampleRate = inFreq;
```

LISTING 4.10 (*Continued*)

```
    if(outFreq > inFreq) maxIOSampleRate = outFreq;
    RATIO = maxIOSampleRate / (float)maxTryFound;
    dec = (float)RATIO * (inFreq / outFreq );
}
cout << "Decimation ratio (dec) = " << dec << endl;
cout << "Interpolation ratio (RATIO) = " << RATIO << endl;

float fp = PERCENT_PASS / ( 200.0f * RATIO );
float fa = ( 200.0f - PERCENT_PASS ) / ( 200.0f * RATIO );
float deltaf = fa - fp;

int nfilt = calcFilterLength( ATT, deltaf, beta );
int lsize = nfilt / RATIO;

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

cout << "Poly phase filter size = " << lsize << endl;

Matrix<float> h( RATIO, lsize );
Vector<float> hist( lsize );
hist = 0.0f;

// Compute Kaiser window sample values
int i = 0;
int j = 0;
float valizb = 1.0f / izero( beta );
float npair_inv = 1.0f / npair;

// n = 0 case
h[i++][j] = 0.0f;
for( int n = 1; n < npair; n++ )
{
    int k = npair - n;
    double alpha = k * npair_inv;
    double y = beta * sqrt( 1.0 - ( alpha * alpha ) );
    double w = valizb * izero( y );
    float ck = RATIO * sin( k * ( PI/RATIO )) / ( k*PI );
    h[i++][j] = w * ck;
    if( i == RATIO )
    {
        i = 0;
        j++;
    }
}
```

LISTING 4.10 (*Continued*)

```
    // Force the pass through point
    h[i][lsize / 2] = 1.0f;

    // Second half of response
    for( n = 1 ; n < npair; n++ )
    {

        // "from" location
        i = npair - n;

        // "to" location
        int k = npair + n;
        h[k % RATIO][k / RATIO] = h[i % RATIO][i / RATIO];
    }

    // Interpolate the data by calls to
    // fir_filter_no_update, decimate the interpolated
    // samples by only generating the samples
    // required
    float phase = 0.0f;

    int lenIn = dspfileIn.getRecLen();
    for( int count = 0; count < lenIn; count++ )
    {
        float signalIn;
        dspfileIn.readElement( signalIn );
        while( phase < (float)RATIO )
        {
            // Pointer to poly phase values
            int k = (int)phase;

            dspfileOut.writeElement(
                filterNoUpdate( signalIn, h.getRow( k ), hist ) );
            phase += dec;
        }
        phase -= RATIO;
        filterUpdate( signalIn, hist );
    }

    // Close files
    dspfileIn.close();
    dspfileOut.close();
}
catch( DSPException& e )
{
```

LISTING 4.10 (*Continued*)

```
      // Display exception
      cerr << e;
      return 1;
  }
  return 0;
}
```

LISTING 4.10 (*Continued*)

```
//////////////////////////////////////////////////////////////////
//
// float izero( float y )
// Compute Bessel function Izero(y) using a
// series approximation.
//
//////////////////////////////////////////////////////////////////
float izero( float y )
{
    float s = 1.0f;
    float ds = 1.0f;
    float d = 0.0f;
    do
    {
        d = d + 2.0;
        ds = ds * ( y * y ) / ( d * d );
        s = s + ds;
    }
    while( ds > 1E-7f * s);
    return s;
}
//////////////////////////////////////////////////////////////////
//
// int calcFilterLength( float att, float deltaf, float &beta )
//   Use att to get beta (for Kaiser window function)
//   and nfilt (always odd valued and = 2 * npair + 1)
//   using Kaiser's empirical formulas.
//
//  Returns:
//   Filter length
//
//////////////////////////////////////////////////////////////////
int calcFilterLength( float att, float deltaf, float &beta )
{
```

LISTING 4.11 Functions **izero, calcFilterLength, filterNoUpdate** and **filterUpdate** used by program PSHIFT.CPP. (*Continued*)

```
    // Value of beta if att < 21
    beta = 0.0f;
    if( att >= 50.0f )
        beta = 0.1102f * ( att - 8.71f );
    if( att < 50.0f && att >= 21.0f )
    {
        beta = 0.5842 * pow( ( att - 21.0f ), 0.4f );
        beta += 0.07886f * ( att - 21.0f );
    }
    int npair = (long int)(( att - 8.0f )/( 28.72f*deltaf ) );
    return( 2 * npair + 1 );
}
////////////////////////////////////////////////////////////////
//
// float filterNoUpdate( input, coef, history )
//   Run the FIR filter using floating point numbers and
//   do not update the history array.
//
//  Returns:
//   Filtered element
//
////////////////////////////////////////////////////////////////
float filterNoUpdate(
    float input,
    const Vector<float> &coef,
    const Vector<float> &history )
{
    int n = coef.length();
    if( n < 2 )
        throw DSPParamException( "Too few coefficients" );

    if( n != history.length() )
        throw DSPParamException( "history does not match" );

    int h = 0;

    // Index to last coef
    int c = n - 1;

    // Form output accumulation
    float output = history[h++] * coef[c—];
    for( int i = 2; i < n; i++ )
        output += history[h++] * coef[c—];

    // Input tap
    output += input * coef[0];
```

LISTING 4.11 (*Continued*)

```
    return(output);
}

////////////////////////////////////////////////////////////////
//
// void filterUpdate( float input, Vector<float> &history )
// Update the fir_filter history array
//
////////////////////////////////////////////////////////////////
void filterUpdate( float input, Vector<float> &history )
{
    int n = history.length();
    if( n < 2 )
        throw DSPParamException( "History buffer too small" );

    // Update history array
    for( int i = 0; i < n - 1; i++ )
        history[i] = history[i+1];

    // Last input
    history[i] = input;
}
```

LISTING 4.11 (*Continued*)

Note that the common factor of 100 was found to allow 441 polyphase filters to be used for the sample rate change. The sampling rate of the CHKL441.DAT file can then be changed to 48000 Hz as follows:

Enter input file : **chkl441.dat**
Enter pitch/sample rate shifted output file name : **chkl480.dat**
Do you want to enter a Musical Key Shift Value (y or n) ? : **n**
Input sample rate [0.01..1e+008] : **44100**
Output sample rate [0.01..1e+008] : **48000**
Max common ratio found (maxTryFound) = 300
Decimation ratio (dec) = 147
Interpolation ratio (RATIO) = 160
Poly phase filter size = 24

Note that the common factor of 300 was found to allow only 160 polyphase filters to be used for this sample rate change.

4.6 COMPLEX FILTERS

Complex signals arise in digital signal processing for several reasons. Sometimes the processing to be performed can be done more efficiently using a complex signal representation. In other cases, the phase or envelope of the signal is of interest and must be extracted from complex data. Perhaps the most fundamental reason for complex signals is that many signals (such as high-frequency radar signals) cannot be sampled directly and must be sampled using two A/D converters and *analog heterodyne circuits* (mixers and lowpass filters). Usually, if the signal can be sampled using a single A/D converter to generate a real sequence, there are some performance advantages. The real signal can then be converted to a complex signal using a Hilbert transform filter (see Section 4.5.1). One performance advantage in using this digital technique over the analog heterodyne approach is that the gain and phase mismatch of the real and imaginary channels in a digital system can be reduced to an arbitrarily small value. Some level of mismatch will always exist in an analog implementation.

In Section 4.2, frequency translation of FIR filters was briefly considered. In most cases, frequency translation of a lowpass FIR filter results in a complex bandpass filter centered at the frequency shift value (shifts from -0.5 to 0.5 are allowed for complex filters) with twice the bandwidth of the lowpass filter. The resulting filter coefficients are complex numbers (see Equation 4.3) and must be filtered with an FIR filter with complex coefficients. However, a complex filter can be realized as four real filters. If x_i, y_i, and g_n are the complex filter input, output and coefficients, respectively, then the complex FIR filter can be expressed as follows:

$$y_i = \sum_{n=0}^{L-1} g_n x_{i-n} \tag{4.16}$$

When the complex multiply is expanded, the following four filters are formed:

$$\mathrm{Re}\{y_i\} = \sum_{n=0}^{L-1} \mathrm{Re}\{g_n\}\mathrm{Re}\{x_{i-n}\} + \sum_{n=0}^{L-1} \mathrm{Im}\{g_n\}\mathrm{Im}\{x_{i-n}\}.$$

$$\mathrm{Im}\{y_i\} = \sum_{n=0}^{L-1} \mathrm{Re}\{g_n\}\mathrm{Im}\{x_{i-n}\} + \sum_{n=0}^{L-1} \mathrm{Im}\{g_n\}\mathrm{Re}\{x_{i-n}\}.$$

where Re { } and Im { } indicate the real and imaginary parts of the argument. Thus, the resulting FIR complex filter can be decomposed into at most four real filters followed by one add and one subtract as shown in Figure 4.21(a). This implementation is generally easier and sometimes more efficient than implementing the complex multiplies directly. In particular, if a complex lowpass filter is implemented and is symmetric about zero frequency, the cross terms (Re { }Im { }) are zero, resulting in only two real filters for the implementation. In general, however, where no symmetry exists or when multiple passbands or stopbands are required, all four filters must be used.

Figure 4.21(b) shows another method to implement some types of complex filters. It is based on frequency translation of the input and output signals so that only two real lowpass filters are required. This technique can be used whenever the desired complex frequency response is symmetric about the selected frequency shift. For example, this method is especially attractive when a very narrow band bandpass filter at a particular complex frequency is required. A lowpass filter is first designed with enough coefficients to have half the desired bandwidth of the bandpass filter. The same filter coefficients can then be used for each of the two filters shown in Figure 4.21(b). The frequency shift can also be made to sweep over a range of frequencies to implement a digital spectrum analyzer. Maximum efficiency can be achieved using two narrow band lowpass IIR filters in this case, since linear phase is generally not required.

Complex filters can also be implemented using the fast convolution techniques as discussed in Chapter 5, Section 5.5. This method is sometimes preferred because most FFT algorithms work very efficiently with complex signals. Because of the wide variety of complex filtering applications and the close association of complex signals with FFT methods, the implementation of real-to-complex conversion using a Hilbert transform filter is the only example considered further in this section.

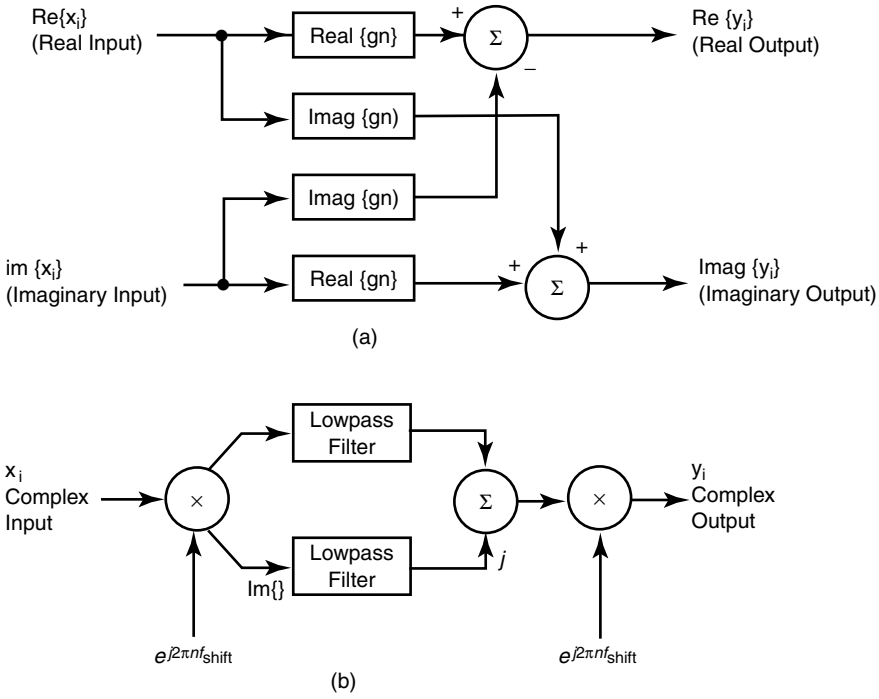


FIGURE 4.21 (a) General complex filter with complex input and complex output decomposed into four real filters. (b) Complex bandpass filter implemented using frequency translation and two real lowpass filters.

4.6.1 Hilbert Transform Real-to-Complex Conversion

A *Hilbert transform filter* is a digital filter that has approximately unity gain and approximately a 90-degree phase shift at all frequencies. It can be used to create an analytic complex signal from a real signal. The analytic signal has a spectrum that is zero (or approximately zero) for all negative frequencies. The analytic complex signal can then be decimated (see Section 4.5) to form a baseband complex signal. Thus, the Hilbert transform filter can be used to convert a real digital sequence with a high sampling rate to a complex digital sequence with a lower sampling rate. The block diagram of this process is shown in Figure 4.22. This real-to-complex conversion structure is a special case of a complex filter where a single real input is used to generate a complex output using two real filters. As long as the input RF signal has limited bandwidth, the digital system shown in Figure 4.22 is essentially equivalent to the mixers and lowpass filters found in analog complex demodulators.

An ideal Hilbert transform filter with unity gain for all frequencies has a noncausal infinitely long impulse response and cannot be implemented. Fortunately, the McClellan-Parks program can be used to generate an optimum FIR Hilbert transform filter with unity gain over a given frequency band with a given number of coefficients. The impulse response and passband frequency response of such a filter with 35 coefficients is shown in Figure 4.23. The passband was specified from 0.02 to 0.48 (a symmetric passband is required). The resulting passband ripple was 1 dB (± 0.5 dB about unity gain). Note that the impulse response shown in Figure 4.23(a) is antisymmetric and has all the odd coefficients equal to zero. The delay of the filter is 17 samples, which must be compensated for by the delay in the real path of the real-to-complex converter shown in Figure 4.22.

Listing 4.12 shows the program REALCMX, which implements the structure illustrated in Figure 4.22 using the 35-point FIR Hilbert transform filter. The **FIRfilter** mem-

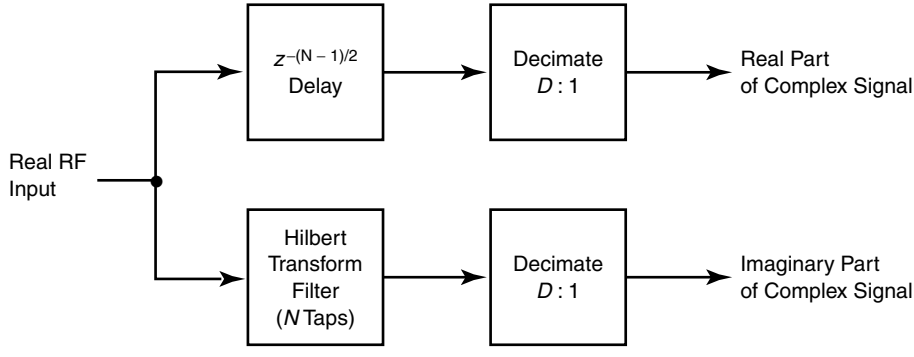
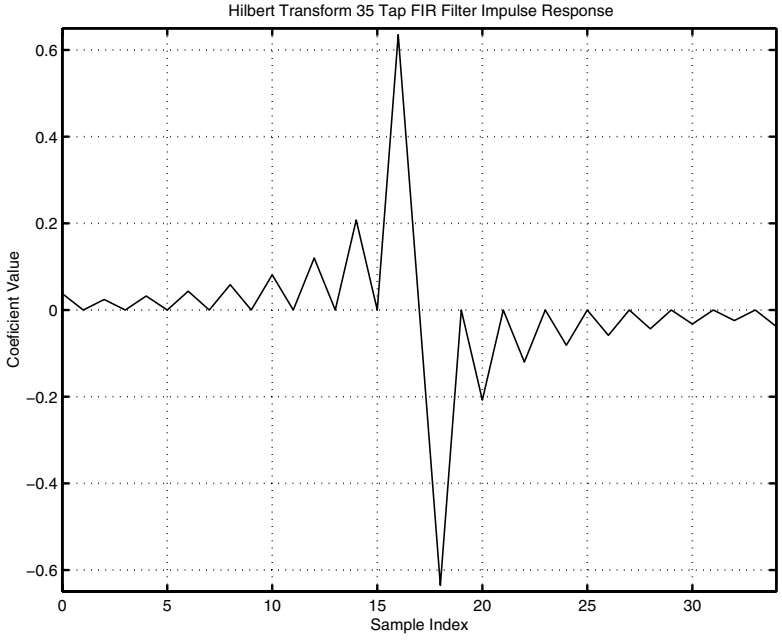
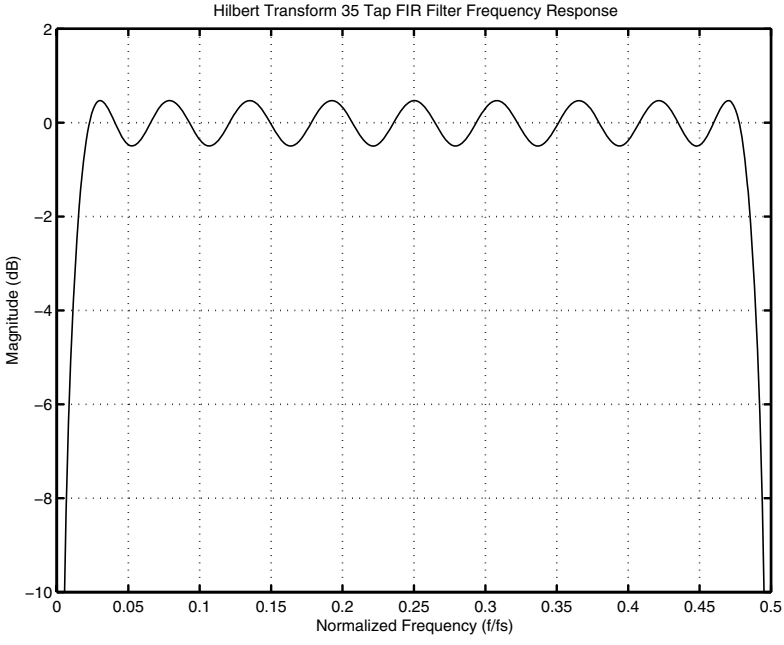


FIGURE 4.22 Digital real-to-complex converter as implemented by REALCMX using a Hilbert transform filter. The input signal is sampled at D times the RF carrier frequency.



(a)



(b)

FIGURE 4.23 (a) Impulse response of 35-point Hilbert transform filter (designed using the McClellan-Parks program). (b) Frequency response magnitude of the 35-point Hilbert transform filter.

```
//////////////////////////////////////////////////////////////////////
//
// realcmx.cpp - Convert RF Data from real to complex
//
//////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "filter.h"

//////////////////////////////////////////////////////////////////////
//
// Constant filter coefficients
//
//////////////////////////////////////////////////////////////////////

// 35 point hilbert transform FIR filter cutoff at 0.02 and 0.48
// +/- 0.5 dB ripple in passband, zeros at 0 and 0.5
float FIRHILBERT35[] =
{
     0.038135F,    0.000000F,    0.024179F,    0.000000F,    0.032403F,
     0.000000F,    0.043301F,    0.000000F,    0.058420F,    0.000000F,
     0.081119F,    0.000000F,    0.120167F,    0.000000F,    0.207859F,
     0.000000F,    0.635163F,    0.000000F,   -0.635163F,    0.000000F,
    -0.207859F,    0.000000F,   -0.120167F,    0.000000F,   -0.081119F,
     0.000000F,   -0.058420F,    0.000000F,   -0.043301F,    0.000000F,
    -0.032403F,    0.000000F,   -0.024179F,    0.000000F,   -0.038135F
};
FIRFilter<float> FIRHILBERT( FIRHILBERT35, ELEMENTS( FIRHILBERT35 ) );

//////////////////////////////////////////////////////////////////////
//
// int main()
//   Converts an RF data record to a DSPFile containing
//   two records representing the real and imaginary part
//   of the signal.
//
//  Returns:
//   0 -- Success
//
//////////////////////////////////////////////////////////////////////
int main()
{
    try
```

LISTING 4.12 Program REALCMX used to convert a DSP data record representing real RF data to two records representing real and imaginary part of the equivalent complex signal. (*Continued*)

```
{
  DSPFile dspfile;
  String strName;
  String strTrailer;

  // Open the input file
  do getInput( "Enter file to change to COMPLEX", strName );
 while( strName.isEmpty() || dspfile.isFound( strName ) == false );
 dspfile.openRead( strName );
 dspfile.getTrailer( strTrailer );

 // Print trailer
 if( strTrailer.isEmpty() == false )
     cout << "File trailer:\n" << strTrailer << endl;

 Vector<float> vIn;
 dspfile.read( vIn );
 dspfile.close();

 // Get type of filter to use of several in filter.h
 int decim = 0;
 getInput( "Enter decimation ratio", decim, 1, 20 );

 // Create output vector
 Vector<float> vOut = FIRHILBERT.filter( vIn );

 // Decimate original input and Hilbert transformed output
 for( int i = 1; i < (vIn.length() + decim - 1)/decim ; i++ )
 {
     vIn[i] = vIn[i * decim];
     vOut[i] = vOut[i * decim];
 }

 // Write filtered vector out to file
 do getInput( "Enter output file name", strName );
 while( strName.isEmpty() );
 dspfile.openWrite( strName );

 // Write two decimated records
 dspfile.write( vIn( 0, vIn.length() / decim ) );
 dspfile.write( vOut( 0, vOut.length() / decim ) );

 // Append to trailer
 char szFmt[100];
 sprintf(
```

LISTING 4.12 (*Continued*)

```
            szFmt,
            "\nConverted to complex using realcmx.cpp, Decimated %d:1",
             decim );
             strTrailer += szFmt;
        dspfile.setTrailer( strTrailer );

        // Close file
        dspfile.close();
    }
    catch( DSPException& e )
    {

        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 4.12 (*Continued*)

ber function (see Section 4.1.1) is used to implement the Hilbert transform filter and automatically compensate for the filter delay. Note that this implementation is somewhat inefficient, since half of the filter coefficients are zero. The result of the REALCMX program is two records representing the real and imaginary parts of the complex signal.

Figure 4.24 shows an example real RF pulse signal with a sampling rate three times the carrier frequency of the pulse. The envelope of the sinewave pulse has a Gaussian-shaped attack and decay (with different time constants) and can be found on the accompanying disk in the file PULSE.DAT. The signal perhaps represents the result that would be obtained by directly sampling a noiseless radar or sonar echo if this were possible. The REALCMX program can be used with this data set as follows:

```
Enter input file name : PULSE.DAT

File trailer:

RF pulse data sampled at 3 times center frequency.

Enter decimation ratio [1...20] : 3

Enter output file name : PULSECMX.DAT
```

The resulting real and imaginary records of the 166 sample complex signal are shown in Figure 4.25. The decimation ratio of 3:1 was chosen so that the resulting complex signal is a baseband representation (centered at zero frequency) of the original RF samples.

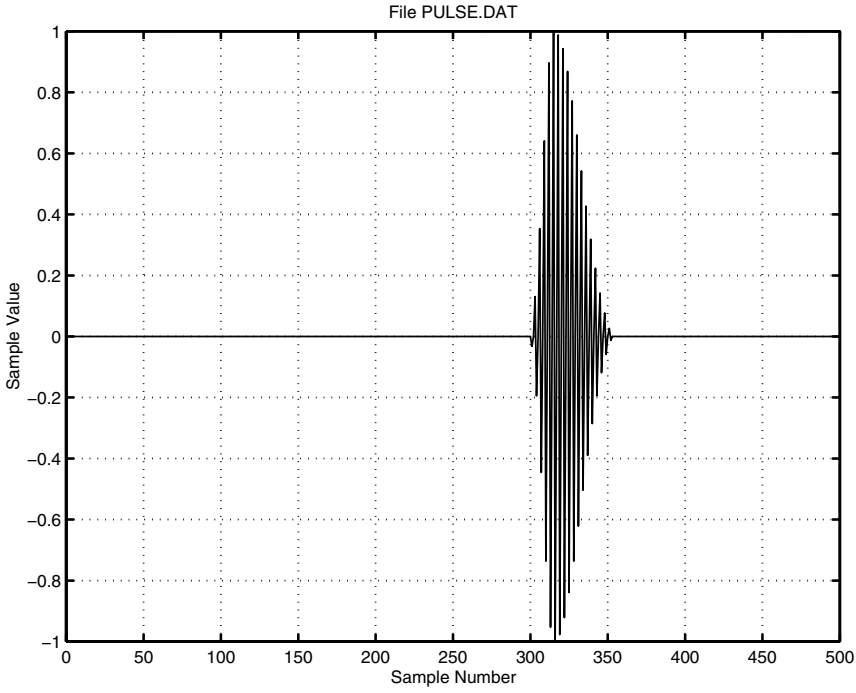
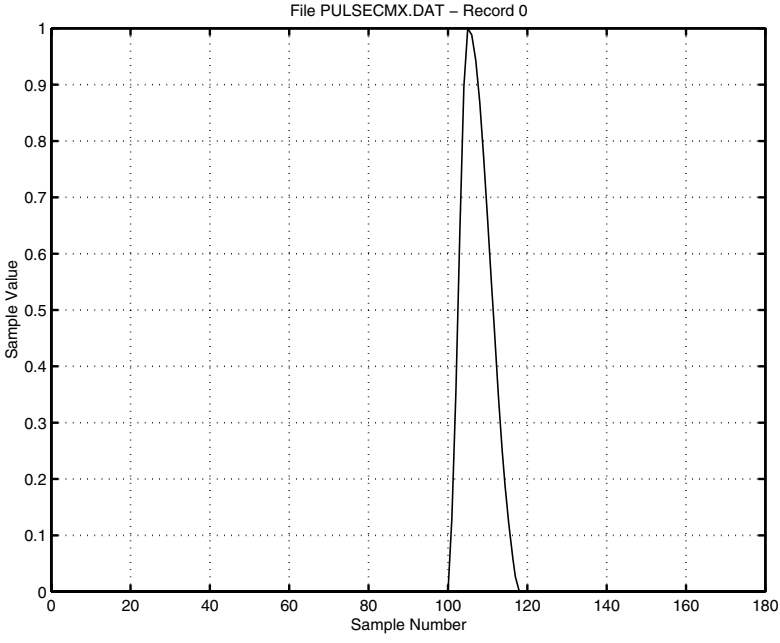


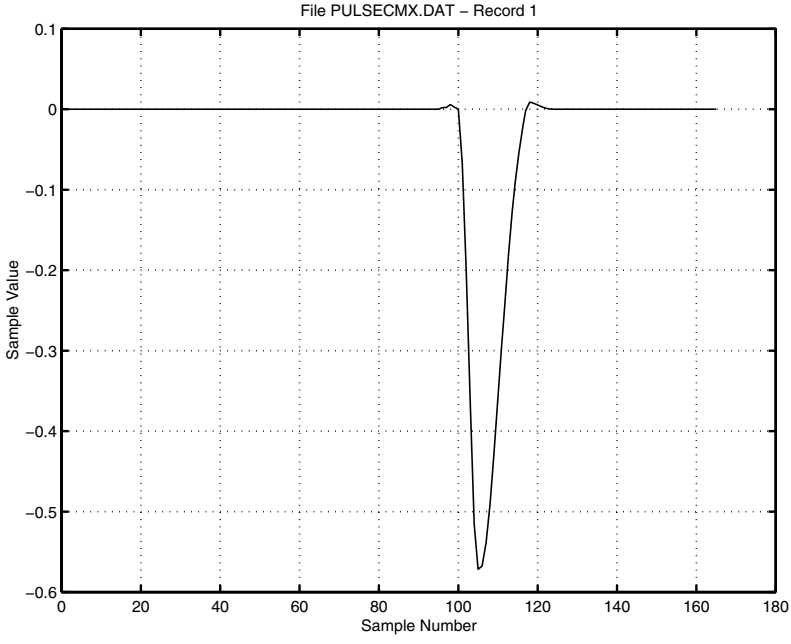
FIGURE 4.24 Plot of data file PULSE.DAT showing a simulated radar or sonar echo. The pulse has a Gaussian attack and decay and a sampling rate of three times the center frequency.

4.7 FILTERING TO REMOVE NOISE

Noise is generally unwanted and can usually be reduced by some type of filtering. Noise can be highly correlated with the signal, or it can be in a completely different frequency band, in which case it is uncorrelated. Some types of noise are impulsive in nature and occur relatively infrequently, while other types of noise appear as narrow band tones near the signal of interest. The most common type of noise is wideband thermal noise, which originates in the sensor or the amplifying electronic circuits. Such noise can often be considered white Gaussian noise implying that the power spectrum is flat and the distribution is normal. The most important consideration in deciding what type of filter to use to remove noise is the type and characteristics of the noise. In many cases, very little is known about the noise process contaminating the digital signal, and it is usually costly (in terms of time and/or money) to find out. Often, the last consideration in a system design is how the noise propagates through the system (the first consideration is often cost or packaging). Unfortunately, the overall signal-to-noise ratio will ultimately determine the system performance.



(a)



(b)

FIGURE 4.25 Plot of the result of converting the PULSE.DAT data file to a complex baseband signal using program REALCMX (a decimation ratio of 3 was used). (a) Real part (record 0). (b) Imaginary part (record 1).

This section describes several examples involving linear filtering to improve the signal-to-noise ratio when white noise is present. Sections 4.7.1 and 4.7.2 describe simple routines to generate and measure noise. These routines will then be used in the noise-filtering examples in Sections 4.7.3 and 4.7.4. The examples are by no means optimum noise reduction examples; they simply attempt to illustrate the basic principles. When impulse like (non-Gaussian) noise is present, nonlinear filtering can sometimes be more effective. Nonlinear filtering is described briefly in Section 4.8.

4.7.1 Noise Generation

The functions **gaussian** and **uniform** (shown in Listing 4.13) are used for noise generation and are contained in the DSP.H source file. Both functions have no arguments and return a single random, double-precision, floating-point number.

The function uniform (shown at the end of Listing 4.13) returns uniform **double** random numbers from -0.5 to 0.5. The standard C library function **rand** is called. The function **rand** (see Appendix A) normally returns integers from 0 to some maximum value (a defined constant, **RAND_MAX,** in ANSI implementations). As shown in Listing 4.13, the integer values returned by **rand** are converted to a **double** value to be returned by **uniform.** Although the random number generator provided with most C compilers gives good random numbers with uniform distributions and long periods, if the random number generator is used in an application that requires truly random, uncorrelated sequences, the generator should be checked carefully. If the **rand** function is in question, a standard random number generator can be easily written in C (see the review article by Park and Miller).

The function **gaussian** (shown in the beginning of Listing 4.13) returns a zero mean random number with a unit variance and a Gaussian (or normal) distribution. It uses the Box-Muller method (see Knuth or Press et al.) to map a pair of independent uniformly distributed random variables to a pair of Gaussian random variables. The function **uniform** is used to generate the two uniform variables that are transformed using the following statements:

```
v1 = 2.*uniform();
v2 = 2.*uniform();
r = v1*v1 + v2*v2;
fac = sqrt(-2.*log(r)/r);
gstore = v1*fac;
gaus = v2*fac;
```

The **r** variable is the radius squared of the random point on the (**v1, v2**) plane. In the **gaussian** function, the **r** value is tested to ensure that it is always less than 1 (which it usually is), so that the region uniformly covered by (**v1, v2**) is a circle, **log(r)** is always negative, and the argument for the square root is positive. The variables **gstore** and **gaus** are the resulting independent Gaussian random variables. Because **gaussian** must return one value at a time, the **gstore** variable is a **static** floating-point variable used to store the

```
////////////////////////////////////////////////////////////////
//
// uniform()
//   Generates zero mean unit uniform random
//   number from -0.5 to 0.5.
//
//  Returns:
//   One zero mean uniformly distributed random number
//
////////////////////////////////////////////////////////////////
inline double uniform()
{
    return( (double)( rand() & RAND_MAX ) / RAND_MAX - 0.5 );
}

////////////////////////////////////////////////////////////////
//
// gaussian()
//   Generates zero mean unit variance Gaussian
//   random numbers. Uses the Box-Muller
//   transformation of two uniform random numbers to
//   Gaussian random numbers.
//
//  Returns:
//   one zero mean unit variance Gaussian random number
//
////////////////////////////////////////////////////////////////
inline double gaussian()
{
    // Flag to indicated stored value
    static bool ready = false;
    // Place to store other value
    static double gstore;
    double v1,v2,r,fac,gaus;

    // Make two numbers if none stored
    if( ready == false )
    {
        do
        {
            v1 = 2.0 * uniform();
            v2 = 2.0 * uniform();
            r = v1 * v1 + v2 * v2;
        }
```

LISTING 4.13 Inline functions **gaussian** and **uniform** (contained in DSP.H) used to generate noise samples. (*Continued*)

```
        // Make radius less than 1
        while( r > 1.0 );

        // Remap v1 and v2 to two Gaussian numbers
        fac = sqrt( 2.0 * -log( r ) / r );
        // Store one
        gstore = v1 * fac;
        // Return one
        gaus = v2 * fac;
        // Set ready flag
        ready = true;
    }
    else
    {
        // Reset ready flag for next pair
        ready = false;
        // Return the stored one
        gaus = gstore;
    }
    return( gaus );
}
```

LISTING 4.13 (*Continued*)

vl*fac result until the next call to **gaussian.** The **static** integer variable **ready** is used as a flag to indicate if **gstore** has just been stored or if two new Gaussian random numbers should be generated.

Listing 4.14 shows the ADDNOISE program, which is used to add noise to the first record in a DSP data file or generate noise only. The ADDNOISE program is normally used to add a sequence of random noise samples to the first record of the input DSP data file and generate a new output file. However, if no input file name is specified (by entering a carriage return), a single record of Gaussian or uniform noise can be generated. The following computer dialogue illustrates the generation of 150 unit variance Gaussian noise samples using the ADDNOISE program:

```
Enter input file name (CR for none) :

Enter number of samples to generate [2...32767] : 150

Enter noise type (0 = uniform, 1 = Gaussian) [0...1] : 1

Enter noise multiplier [0...1e+003] : 1

Enter output file name : WNG.DAT

Noise signal of length 150, Gaussian distribution, Amplitude = 1.0
```

```
//////////////////////////////////////////////////////////////////////////
//
// addnoise.cpp - Add noise to DSPFile
//   Program to add noise to existing DSPFile or create
//   noise only. Adds noise to first record of input file
//   and generates a one record output file.
//
//  Inputs:
//   Filename and length, type, and amplitude of noise
//
//  Outputs:
//   DSPFile with noise added
//
//////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
int main()
{
    try
    {
        DSPFile dspfile;
        String strFile;
        String strTrailer;
        int i = 0;

        // Get parameters from user
        do getInput( "Enter file name", strFile );
        while( strFile.isEmpty() );

        const int UNIFORM = 0;
        const int GAUSSIAN = 1;

        int tn = 0;
        getInput(
            "Type of noise ( 0 = uniform, 1 = Gaussian )",
            tn,
            UNIFORM,
            GAUSSIAN );

        float noiseMult = 0.0f;
        getInput("Enter noise multiplier",noiseMult,0.0f, 1000.0f);

        // Vector containing noise
        Vector<float> vecNoise;
```

LISTING 4.14 Program ADDNOISE used to add Gaussian or uniform white noise to a DSP data file record (or generate noise alone). (*Continued*)

```
        // Check if data is in file or should be generated
        if( dspfile.isFound( strFile ) )
        {
            dspfile.openRead( strFile );
            dspfile.getTrailer( strTrailer );

            // Read float vector from existing file
            dspfile.read( vecNoise );
            dspfile.close();

            // Get output file from user
            do getInput( "Enter output file name", strFile );
            while( strFile.isEmpty() );
        }
        else
        {
            // Generate data
            int len = 0;
            getInput("Enter number of samples to generate", len,2,32767);

            // Allocate vector to len and set all elements to 0.0
            vecNoise.setLength( len ) = 0.0f;
        }

        // Add noise to record
        if( tn == UNIFORM )
        {
            for( i = 0; i < vecNoise.length(); i++)
                vecNoise[i] += (float)(noiseMult * uniform() );
        }
        else
        {
            for( i = 0; i < vecNoise.length(); i++)
                vecNoise[i] += (float)(noiseMult * gaussian() );
        }

        // Write out new data
        dspfile.openWrite( strFile );
        dspfile.write( vecNoise );

        // Make descriptive trailer and write to file
        char fmtBuf[80];
        sprintf(
            fmtBuf,
            "Noise signal of length %d, %s distribution, Amplitude = %f\n",
```

LISTING 4.14 (*Continued*)

```
            vecNoise.length(),
            ( tn == UNIFORM ) ? "Uniform" : "Gaussian",
            (float)noiseMult);
        // Append to trailer
        strTrailer += fmtBuf;
        dspfile.setTrailer( strTrailer );

        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

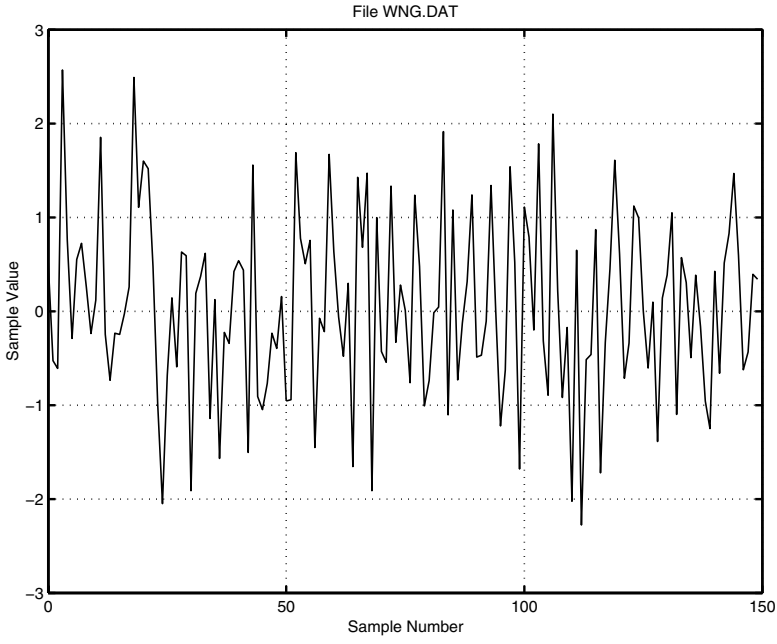
LISTING 4.14 (*Continued*)

Figure 4.26(a) shows the resulting 150 Gaussian noise samples from the ADDNOISE program. Although different implementations of the C library function **rand** will result in different random sequences, Figure 4.26(a) is representative of Gaussian white noise. Different random sequences can be generated by using the **srand** function to change the seed used by the **rand** function. Figure 4.26(b) shows the result of using the ADDNOISE program when the uniform noise type is selected (noise type = 0). The uniform sequence has far more samples near the extremes than the Gaussian random sequence as expected.

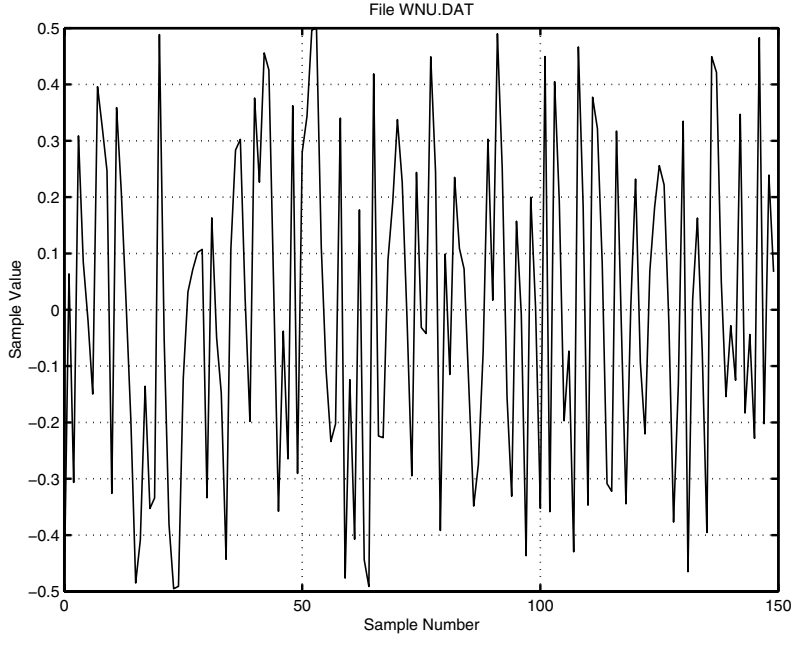
4.7.2 Statistics Calculation

The graphic display of the records in a DSP data file obtained using the WINPLOT program is very useful for examining the qualitative aspects of the data. However, in many cases it is important to extract quantitative information from the data in order to compare the performance of different signal processing algorithms. The STATS program shown in Listing 4.15 provides the basic quantitative statistics of each record in a DSP data file. The minimum value, maximum value, average, variance, and standard deviation (denoted by sigma) of each record are displayed in a table format.

The STATS program uses the **read** function to read in each record of the DSP data file. Four functions (**average**, **variance**, **minimum**, and **maximum**) are then used to calculate the individual statistics of each record read. The statistics functions are similar to the statistics functions shown in Listing 2.1 and discussed in Section 2.1 of Chapter 2.



(a)



(b)

FIGURE 4.26 Plot of 150 white noise samples generated by program ADDNOISE. (a) Gaussian distributed. (b) Uniformly distributed.

```
//////////////////////////////////////////////////////////////////
//
// stats.cpp - Calculate standard statistics on DSPFiles
//   Finds min, max, mean, and variance of
//   records in an existing DSPFile.
//   Calls minimum, maximum, average, and variance
//   functions contained in this file.
//
//  Inputs:
//   Input file name
//
//  Outputs:
//   Statistics of each record in input file
//
//////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
//////////////////////////////////////////////////////////////////
//
// minimum( const Vector<Type>& v )
//
//  Return:
//   Minimum value in vector
//
//////////////////////////////////////////////////////////////////
template <class Type>
Type minimum( const Vector<Type>& v )
{
    int len = v.length();
    if( len == 0 )
        return 0;

    Type ans = v[0];
    for( int i = 1; i < len; i++ )
    {
        if( v[i] < ans )
            ans = v[i];
    }
    return( ans );
}
//////////////////////////////////////////////////////////////////
//
```

LISTING 4.15 Program STATS used to determine the min, max, mean, variance, and standard deviation of each record in a DSP data file. (*Continued*)

```
// maximum( const Vector<Type>& v )
//
//  Return:
//   Maximum value in vector
//
////////////////////////////////////////////////////////////////
template <class Type>
Type maximum( const Vector<Type>& v )
{
    int len = v.length();
    if( len == 0 )
        return 0;

    Type ans = v[0];
    for( int i = 1; i < len; i++ )
    {
        if( v[i] > ans )
            ans = v[i];
    }
    return( ans );
}
////////////////////////////////////////////////////////////////
//
// average( const Vector<Type>& v )
//
//  Return:
//   Average of all values in vector
//
////////////////////////////////////////////////////////////////
template <class Type>
Type average( const Vector<Type>& v )
{
    int len = v.length();
    if( len == 0 )
        return 0;
    return( sum( v ) / len );
}
////////////////////////////////////////////////////////////////
//
// variance( const Vector<Type>& v )
//
//  Return:
//   Variance in vector
//
////////////////////////////////////////////////////////////////
```

LISTING 4.15 (*Continued*)

```
template <class Type>
Type variance( const Vector<Type>& v )
{
    int len = v.length();
    if( len == 0 )
        return 0;

    Type sum = (Type)0;
    Type sum2 = (Type)0;
    for( int i = 0; i < len; i++ )
    {
        Type element = v[i];
        sum += element;
        sum2 += element * element;
    }

    Type ave = sum / len;
    return( ( sum2 - sum * ave ) / ( len - 1 ) );
}
//////////////////////////////////////////////////////////////////
//
// int main()
//   Find statistics of an existing DSPFile.
//
//  Returns:
//   0 -- Success
//
//////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String str;

        // Open the file
        do getInput( "Enter input file name", str );
        while( str.isEmpty() || dspfile.isFound(str) == false );
        dspfile.openRead( str );

        // Get trailer
        dspfile.getTrailer( str );

        // Display current trailer
        if( str.isEmpty() == false )
```

LISTING 4.15 (*Continued*)

```
            cout << "Current trailer:\n" << str << endl;

        // Display format of data
        cout << " Record Average Variance Sigma   Min   Max\n";
        cout << " ------ ------- -------- -----   ---   ---\n";

        // For each record in file
        for( int i = 0; i < dspfile.getNumRecords(); i++ )
        {
            Vector<float> v;

            // Read any type into float vector
            dspfile.read( v );

            // Format output
            float var = variance( v );
            char fmtBuf[80];
            sprintf(
                fmtBuf,
                "%5d %12.4f %9.4f %9.4f %9.4f %9.4f\n",
                i,
                average( v ),
                var,
                sqrt( var ),
                minimum( v ),
                maximum( v ) );
            cout << fmtBuf;
        }
        cout << endl;
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 4.15 (*Continued*)

The single-record Gaussian noise data shown in Figure 4.26(a) can be analyzed using the STATS program as follows:

```
Enter input filename: WNG.DAT
File trailer:
Noise signal of length 150, Gaussian distribution, Amplitude = 1.0
Record  Average Variance Sigma    Min    Max
------  ------- -------- -----    ---    ---
   0    0.0575   0.9449  0.9720 -2.2759 2.5706
```

The uniform noise data shown in Figure 4.26b gives a result from the STATS program as follows:

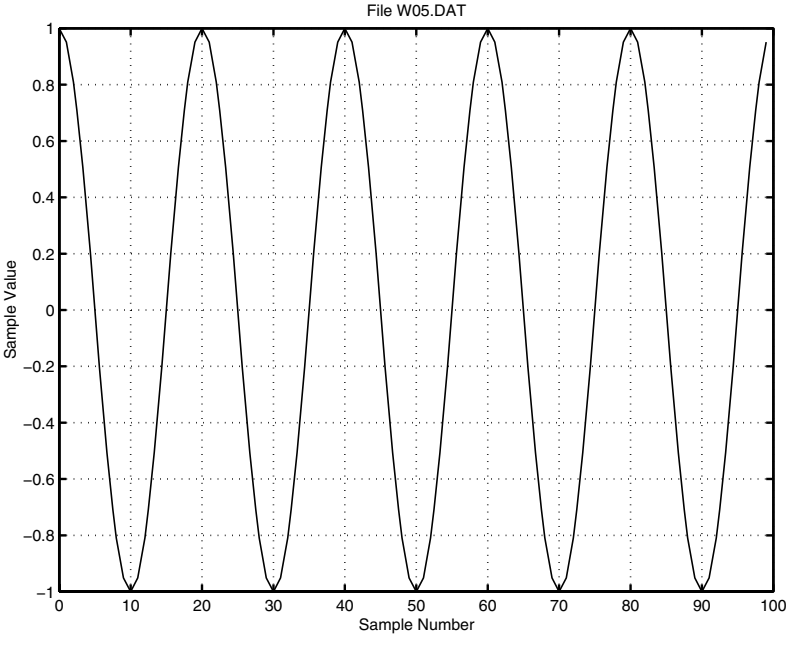
```
Enter input filename: WNU.DAT
File trailer:
Noise signal of length 150, Uniform distribution, Amplitude = 1.0
Record  Average Variance Sigma     Min    Max
------  ------- -------- -----     ---    ---
 0      0.0021  0.0809   0.2845 -0.4987 0.4997
```

Note that the variance and standard deviation of the Gaussian noise is very close to unity as was specified when the ADDNOISE program was used to generate the data. The variance of the uniform random sequence is 0.0809, which is close to the theoretical value of 1/12.

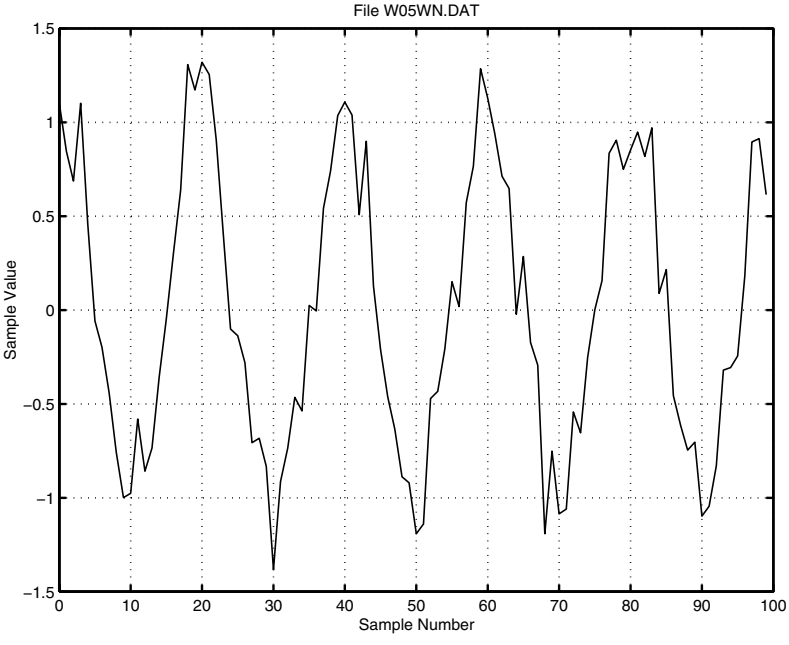
4.7.3 Signal-to-Noise Ratio Improvement

One common application of digital filtering is *signal-to-noise ratio enhancement.* If the signal has a limited bandwidth and the noise has a broad spectrum, then a filter can be used to remove the part of the noise spectrum that does not overlap the signal spectrum. If the filter is designed to match the signal perfectly so that the maximum amount of noise is removed, then the filter is called a *matched* or *Wiener filter.* Wiener filtering is briefly discussed in Section 1.3.5 of Chapter 1.

Figure 4.27 shows a simple example of filtering a single tone with added white noise. Figure 4.27(a) shows the original cosine wave at a 0.05 f_s frequency generated using the MKWAVE program. The ADDNOISE program was then used to add Gaussian white noise with a standard deviation of 0.2 to the cosine wave as shown in Figure 4.27(b). Using the STATS program, the standard deviation of the signal alone [Figure 4.27(a)] can be easily found to be 0.7107. Because the standard deviation of the added noise is 0.2, the signal-to-noise ratio of the noisy signal shown in Figure 4.27(b) is 3.55 or 11.0 dB. Figure 4.27(c) shows the result of applying the 35-tap lowpass FIR filter to the noisy signal. Note that much of the noise is still present but is smaller and has pre-

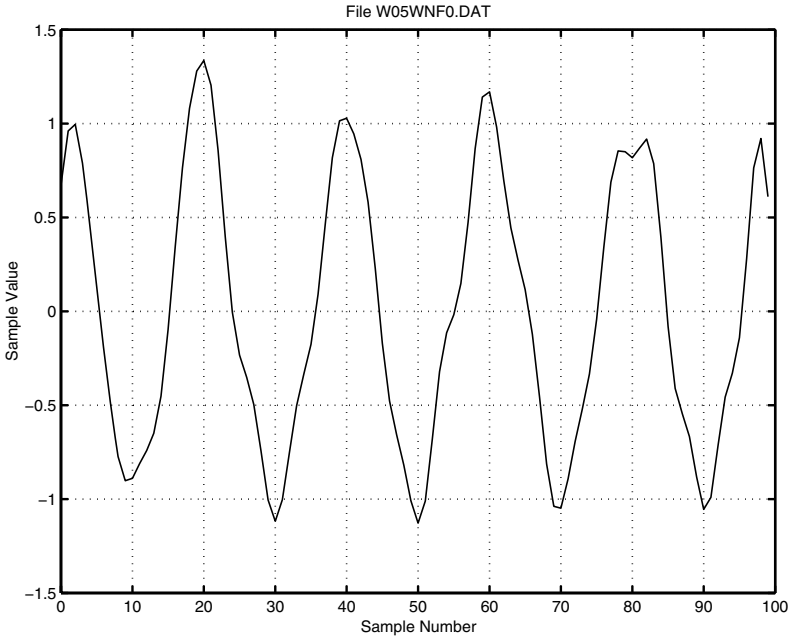


(a)



(b)

FIGURE 4.27 Plot of cosine wave with added noise generated by the MK-WAVE and ADDNOISE programs (frequency = 0.05). (a) Original without noise. (b) Unfiltered version with added Gaussian noise (standard deviation = 0.2). (c) Lowpass filter with 35-point FIR filter.



(c)

FIGURE 4.27 (*Continued*)

dominantly low frequency components. By lowpass filtering the 100 noise samples added to the cosine wave separately, the signal-to-noise ratio of Figure 4.27(c) can be estimated. The following computer dialogue involving the ADDNOISE, STATS, and FIRFILT programs illustrates this procedure:

```
>ADDNOISE

Enter input file name (CR for none):
Enter number of samples to generate [2...32767] : 100
Enter noise type (0 = uniform, 1 = Gaussian) [0...1] : 1
Enter noise multiplier [0...1e + 003] : 0.2
Enter output file name: WN100.DAT
Noise signal of length 100, Gaussian distribution, Amplitude = 0.2

>STATS
Enter input filename : WN100.DAT
File trailer:
Noise signal of length 100, Gaussian distribution, Amplitude = 0.2
Record  Average  Variance Sigma     Min    Max
------  -------  -------- -----     ---    ---
0       0.0146   0.0389   0.1973  -0.4098  0.5141
```

```
>FIRFILT
Enter input file name : WN100.DAT
File trailer:
Noise signal of length 100, Gaussian distribution, Amplitude = 0.2
Enter filter type (0 = LPF, 1 = HPF, 2 = PULSE) [0...2] : 0
Enter output file name : WN100F0.DAT
>STATS
Enter input filename : WN100F0.DAT
File trailer:
Noise signal of length 100, Gaussian distribution, Amplitude = 0.2
Lowpass filtered using FIRFILT

Record Average Variance Sigma     Min     Max
------ ------- -------- -----     ---     ---
0      0.0160  0.0158   0.1257  -0.3093  0.3480
```

As shown above, the standard deviation of the noise alone in Figure 4.27(b) is 0.1973 (well within statistical limits of the 0.2 value selected), and in Figure 4.27(c) it is reduced by the lowpass filter to 0.1257. Because the cosine wave signal level is unchanged in amplitude, the signal-to-noise ratio of Figure 4.27(c) is 5.65 or 15 dB. Thus, the filtering operation improved the signal-to-noise ratio by a factor of 0.637 or 3.9 dB.

The ratio of the output noise variance to the input noise variance of a lowpass filter driven by white Gaussian noise can also be determined analytically as follows:

$$\frac{\sigma_{\text{out}}^2}{\sigma_{\text{in}}^2} = \frac{\int_0^{f_s/2} |H(f)|^2 \, df}{|H(0)|^2} = \frac{\sum_{n=0}^{N-1} b_n^2}{\left| \sum_{n=0}^{N-1} b_n \right|^2} \tag{4.17}$$

The filter frequency response is $H(f)$, and the impulse response of the N tap FIR filter is b_n. $H(0)$ is the maximum gain of the filter in the passband (assumed to be at zero frequency for a lowpass filter). Note that the lower the cutoff frequency of the filter, the greater the noise variance is reduced. The above calculation for the 35-tap FIR lowpass filter yields 0.4079. The standard deviation of the noise is reduced by the square root of this, or 0.6387, in good agreement with the result obtained with the ADDNOISE, FIRFILT, and STATS programs. Although the analytic method works quite well for Gaussian noise, it is often easier to simulate the filtering of non-Gaussian or narrowband correlated noise samples and determine the resulting statistics rather than derive the analytic solution.

Another example of signal-to-noise improvement arises in the detection and parameter estimation of radar or sonar pulses that have been corrupted by noise. By removing some of the noise without changing the signal spectrum, the detection of a radar return becomes much easier and parameters estimated from the signal (such as range to the target) become much more precise. Figure 4.28 shows a very noisy version of the simulated

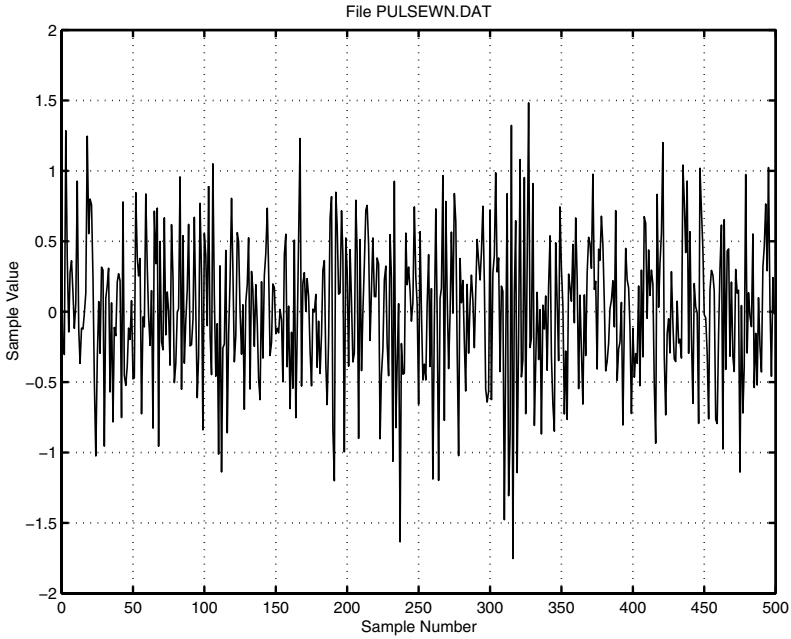
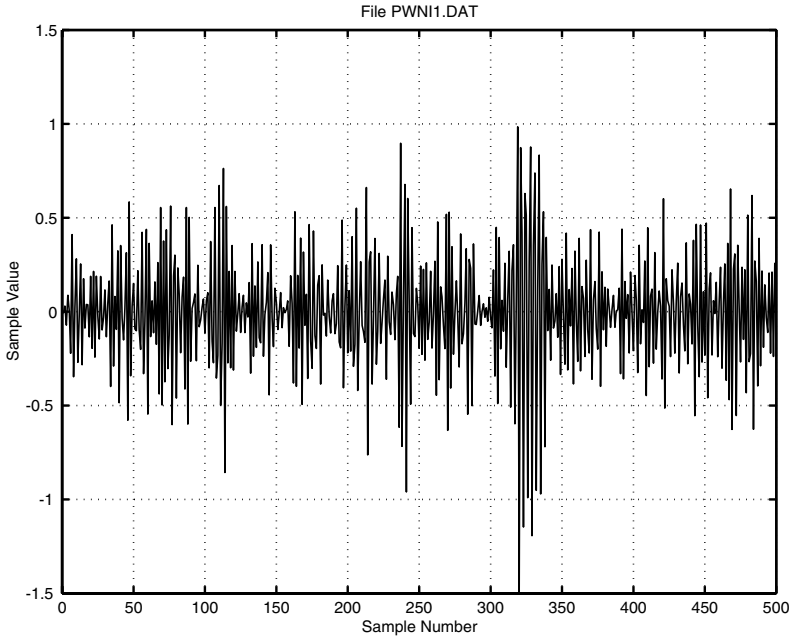


FIGURE 4.28 Plot of data file PULSE.DAT (see Figure 4.24) with added noise (standard deviation of noise = 0.5).

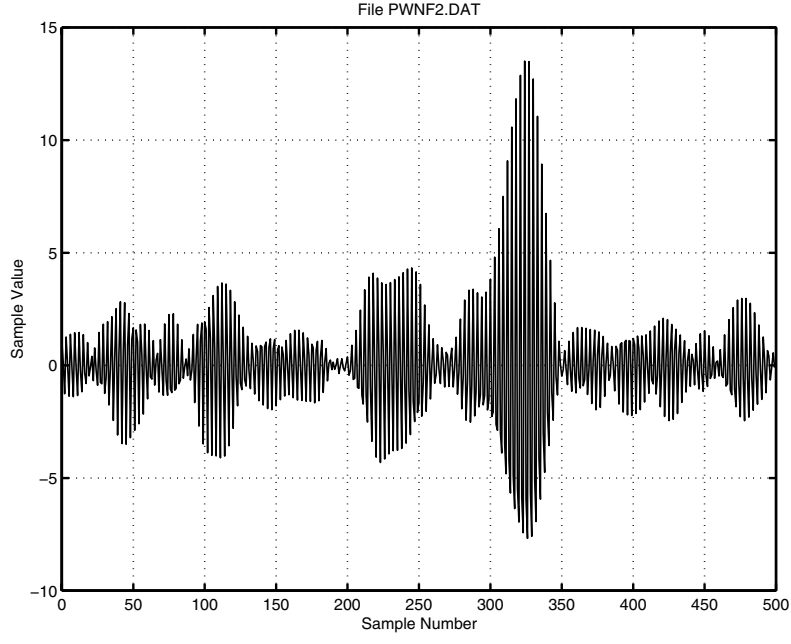
radar pulse first introduced in Section 4.6.1 (see Figure 4.24 for the original waveform without noise). The standard deviation of the white Gaussian noise added by the ADDNOISE program is 0.5. Because the standard deviation of the pulse is also 0.5 (calculated based on the pulse length of 52 samples), Figure 4.28 illustrates a 0-dB signal-to-noise ratio, which makes it very difficult to identify the location of the pulse.

Filtering the noisy RADAR pulse signal will improve the detectability of the pulse because the 3dB bandwidth of the pulse signal is approximately 5% of the bandwidth of the white noise. The center frequency of the pulse is at $0.33f_s$, so a highpass filter that passes the pulse and attenuates some of the low-frequency noise will improve the signal-to-noise ratio. For example, Figure 4.29(a) shows the result of filtering the noisy pulse data with the sixth order Chebyshev IIR highpass filter (described in Section 4.2.2). It is somewhat easier to distinguish the pulse from the noise in Figure 4.29(a) than in Figure 4.28.

A better solution to enhancing the noisy radar pulse data is to perform matched filtering. *Matched filtering* can provide the maximum signal-to-noise ratio for a given pulse spectrum and noise spectrum. The detection probability and the precision of the parameter estimates can be maximized in many cases (see Papoulis for the conditions for maximum likelihood signal detection). When the noise spectrum is white, the matched filter turns out to be a filter with a frequency response equal to the complex conjugate of the pulse spectrum. In the time domain, this implies that the impulse response is the time-



(a)



(b)

FIGURE 4.29 Plot of data file PULSE.DAT with added noise after filtering using IIRFILT and FIRFILT. (a) Sixth-order IRR highpass filter. (b) Matched 52-point FIR filter.

reversed pulse without noise. The 52-point FIR filter defined in FILTER.H and implemented in program FIRFILT is the matched filter for the pulse shown in Figure 4.24. The output of this matched filter when applied to the noisy pulse is shown in Figure 4.29(b). When compared with the highpass filter result shown in Figure 4.29(a), the pulse is much easier to locate and would be easy to detect using a simple amplitude threshold detector.

The signal-to-noise ratio improvement of the matched filter and highpass filter can be estimated in a quantitative way using the ADDNOISE, IIRFILT, FIRFILT, and STATS programs. For each filter, the filter response (in terms of standard deviation) due to the signal alone and the noise alone is estimated. Because each filter is linear and the original signal-to-noise ratio was 0 dB, the ratio of the standard deviation is the improvement in the signal-to-noise ratio. The standard deviation results of this procedure for the noisy radar pulse data are summarized in the following table:

	Signal Only (52 points)	Noise Only (500 points)	Signal-to-Noise Ratio (dB)
Without filter	0.506	0.490	0.3
Sixth-order IIR filter	0.478	0.289	4.4
52-tap matched filter	6.636	1.696	11.8

Thus, the matched FIR filter results in a 7.4 dB better signal-to-noise ratio than the sixth-order IIR filter and is 11.5 dB better than no filter at all. Note that the matched filter has a gain of 13 because no scaling was performed when the pulse signal was time reversed.

4.7.4 Filtering Quantization Noise

In many signal processing systems, the number of bits available to represent the input signal is limited because of the high cost of precision A/D converters or because a high-speed A/D converter is only available with a limited number of bits. In either case, it is possible to design a digital filter tp improve the effective number of bits and the signal-to-noise ratio of the digital representation of the input signal. In practice, the quantization noise present in a signal can often be considered white noise as long as the sampling rate is less than 10 times the dominant frequency and the number of bits is greater than 4. Thus, in some cases filtering quantization noise is essentially the same as filtering white noise as discussed in Section 4.7.3. If the least significant bit of a digital sequence is assigned unit value, then the quantization noise created by the digital representation has a variance of 1/12 and is uniformly distributed (see Chapter 1, Section 1.7.6). Each factor of two improvement in the signal-to-noise ratio (as calculated for a particular filter using Equation 4.17) gives another bit of valid signal information.

Listing 4.16 shows the program QUANTIZE, which simulates the quantization process by rounding the data from a DSP data file to a maximum integer value. For two's-complement rounding, the maximum integer output value (entered by the user) should be set to $2^{B-1} - 1$, where B is the number of bits ($B > 1$). The QUANTIZE pro-

```
////////////////////////////////////////////////////////////////////////
//
// quantize.cpp - Quantize record of DSP data
//
////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "image.h"
#include "get.h"
////////////////////////////////////////////////////////////////////////
//
// int main()
//   Program to quantize a record in a DSPFile so
//   that the maximum absolute value is an integer
//   entered by the user. The resulting DSPFile
//   has the quantized data and the error of the
//   quantization as two records.
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Open the input file
        do getInput( "Enter input file name", strName );
        while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        Vector<float> vIn;
        dspfile.read( vIn );

        // Check if file is empty
        if( vIn.isEmpty() )
            throw DSPFileException( "File empty" );

        // Get file trailer
        String strTrailer;
```

LISTING 4.16 Program QUANTIZE used to determine the integer values of a quantized representation of a data record in a DSP data file. The program also generates a second record showing the quantization error of the signal. (*Continued*)

```
dspfile.getTrailer( strTrailer );
dspfile.close();

// Show trailer
if( strTrailer.isEmpty() == false )
    cout << "File trailer: " << strTrailer << endl;

// Find maximum absolute value
float max = fabs( vIn[0] );
for( int i = 1; i < vIn.length(); i++ )
{
    float x = fabs( vIn[i] );
    if( x > max )
        max = x;
}

cout << "Signal absolute max = " << max << endl;

// Calculate error
Vector<float> vErr( vIn.length() );
int maxInt = 0;
getInput("Enter max absolute integer value",maxInt,1,32767);

// Quantize record and calculate error vector
for( i = 0; i < vIn.length(); i++ )
{
    float x = (maxInt * vIn[i] ) / max;
    float sig = (float)round( x );
    vIn[i] = sig;
    vErr[i] = x - sig;
}

// Open output DSPFile
do getInput( "Enter output file name", strName );
while( strName.isEmpty() );

dspfile.openWrite( strName );
dspfile.write( vIn );
dspfile.write( vErr );

// Make descriptive trailer and write to file
char szBuf[300];
sprintf(
 szBuf,
 "Quantized signal and error\nSignal max = %f, Int max = %d\n",
```

LISTING 4.16 (*Continued*)

```
            max,
            maxInt );
        strTrailer += szBuf;
        dspfile.setTrailer( strTrailer );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        cerr << e;
        return 1;
    }
    return 0;
}
```

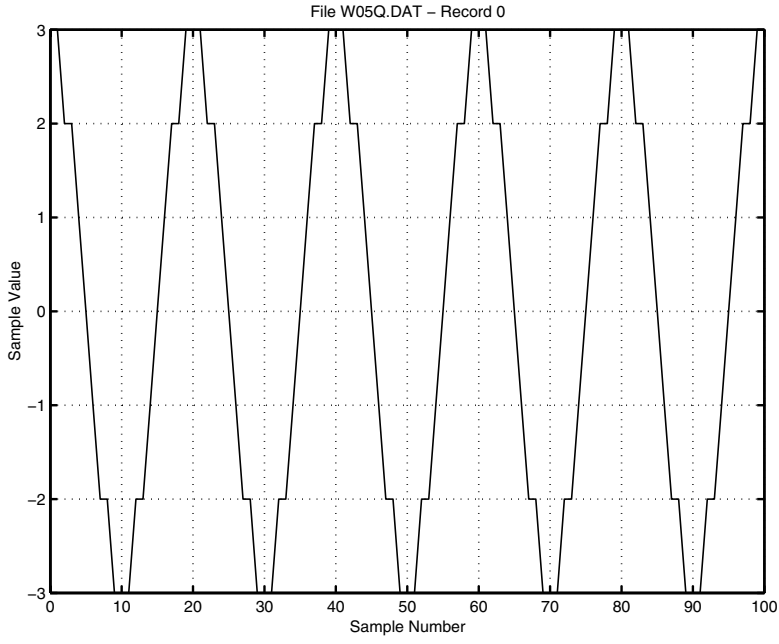
LISTING 4.16 (*Continued*)

gram uses the **round** macro (defined in GET.H; see Chapter 2, Section 2.6.2) to determine the positive or negative integer closest to the scaled floating-point value read by **read.** Only the first data record in the DSP data file is quantized. The program also generates the quantization error record that is the difference between the output and the exact representation. This error record is always between -0.5 and 0.5 and is stored in the second record (record 1) in the output file.

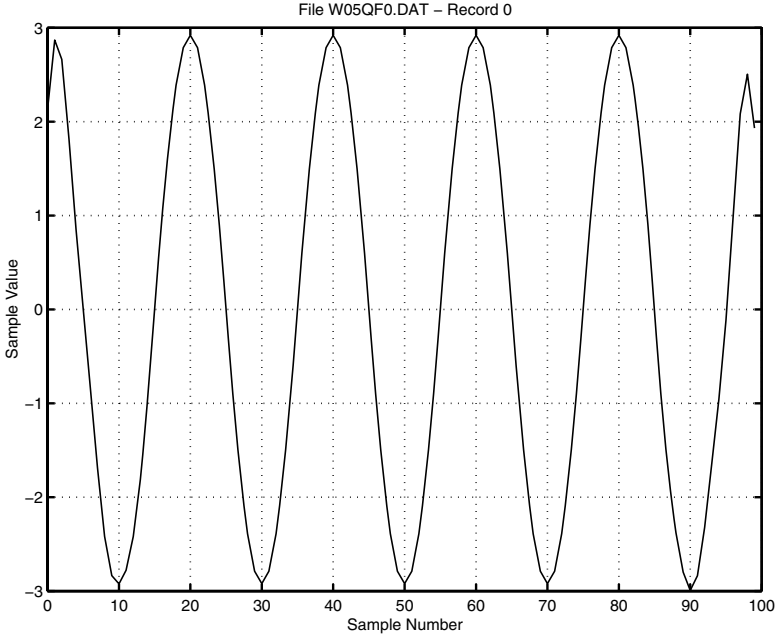
Figure 4.30(a) shows the quantization of a cosine wave generated using the MKWAVE and QUANTIZE programs. The original floating-point data (file W05.DAT) is a single frequency at $0.05f_s$ and is shown in Figure 4.27(a). The computer dialogue required to quantize file W05.DAT using the QUANTIZE program is as follows:

```
Enter input file name : W05.DAT

File trailer:

Signal of length 100 equal to the sum of 1 cosine waves at the following frequencies:
f/fs = 0.050000

Signal absolute max = 1.000000

Enter max absolute integer value [1...32767] : 3

Enter output file name : W05Q.DAT
```

By using the FIRFILT program to filter the 3-bit quantized cosine wave with the 35-tap FIR lowpass filter the result shown in Figure 4.30(b) is obtained. Because the QUANTIZE program provides the quantization error vector as the second record (record



(a)



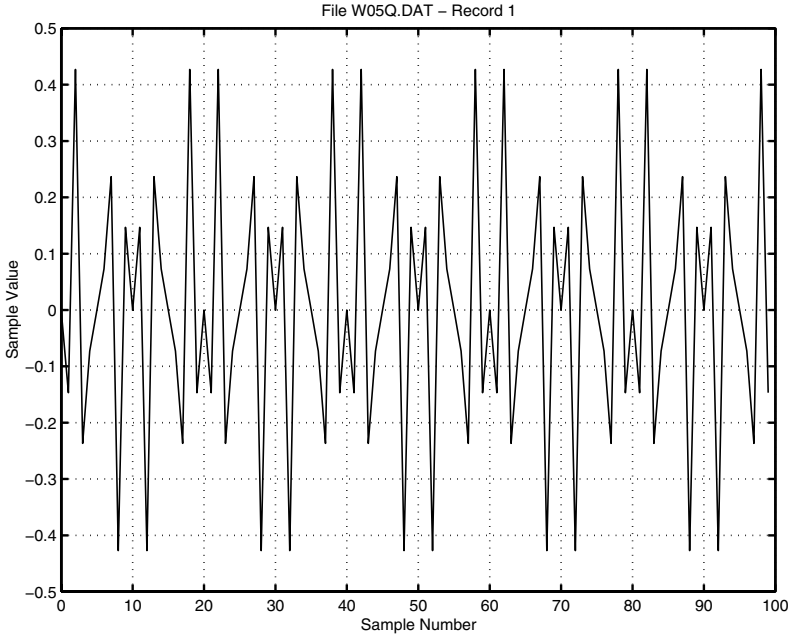
(b)

FIGURE 4.30 Illustration of filtering a quantized signal. (a) Example 3-bit quantized version of W05.DAT. (b) 35-tap lowpass FIR filtered version of the 3-bit quantized W05.DAT.

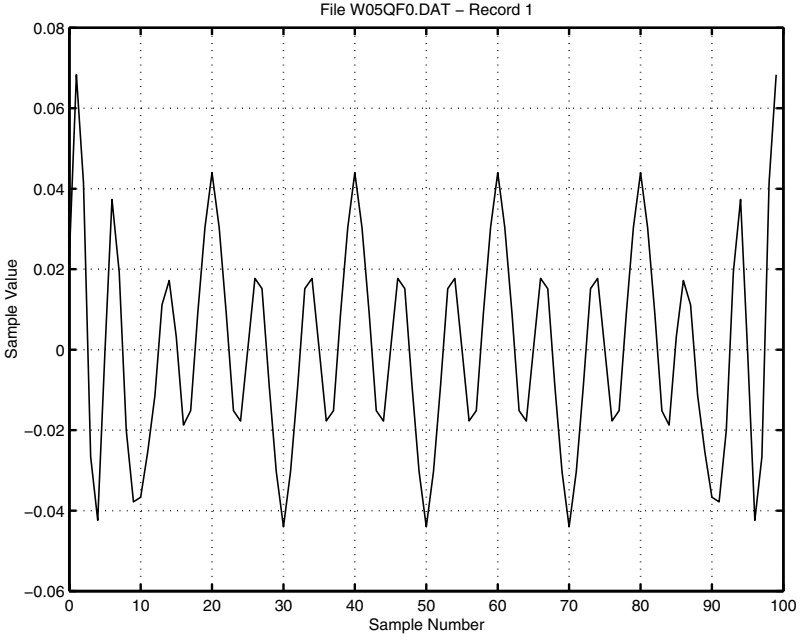
1) in the output file, the quantization error alone is also filtered by the FIRFILT program. Figure 4.31 shows the quantization error vector before and after filtering [record 1 of files W05Q.DAT and W05QF0.DAT are shown in Figures 4.31(a) and 4.31(b), respectively]. Since the filtering is linear and obeys superposition, the filtering of the quantization error alone also shows the affect of the filtering on the quantization noise in the signal. The STATS program can then be used to measure the changes in quantization noise as follows:

```
>STATS

Enter input filename : W05Q.DAT

File trailer:

Signal of length 100 equal to the sum of 1 cosine waves at frequencies:
f/fs = 0.050000

Quantized signal and error
Signal max = 1.000000, Int max = 3

Record  Average  Variance  Sigma    Min    Max
------  -------  --------  -----    ---    ---
   0    0.0000    4.5455  2.1320 -3.0000 3.0000
   1    0.0000    0.0536  0.2315 -0.4271 0.4271

>STATS

Enter input filename: W05QFO.DAT

File trailer:

Signal of length 100 equal to the sum of 1 cosine waves at frequencies:
f/fs = 0.050000

Quantized signal and error
Signal max = 1.000000, Int max = 3

Lowpass filtered using FIRFILT

Record  Average  Variance  Sigma      Min     Max
------  -------  --------  -----      ---     ---
   0   -0.0098   4.3514    2.0860  -2.9924  2.9202
   1    0.0007   0.0007    0.0258  -0.0440  0.0683
```

The signal-to-noise ratio of the quantized signal shown in Figure 4.30(a) is 19.3 dB (2.132/0.2315 as shown for W05Q.DAT), which is very close to the theoretical value of



(a)



(b)

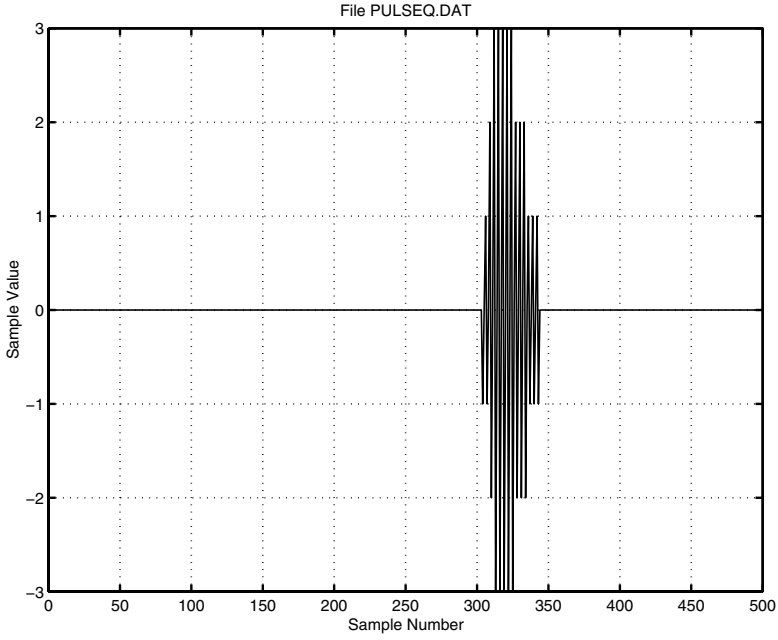
FIGURE 4.31 Illustration of filtering the quantization error of the signal shown in Figure 4.30(a). (a) The error of the 3-bit quantized cosine wave. (b) 35-tap lowpass FIR filtered version of the quantization error.

19.6 dB (see Chapter 1, Section 1.7.6). The signal-to-noise ratio of the lowpass filtered version is 38.2 dB. This is much greater than the expected value of 23.5 dB (a 3.9-dB improvement) if the quantization noise was independent white noise. Unfortunately, in some cases, quantization noise is not independent or white.

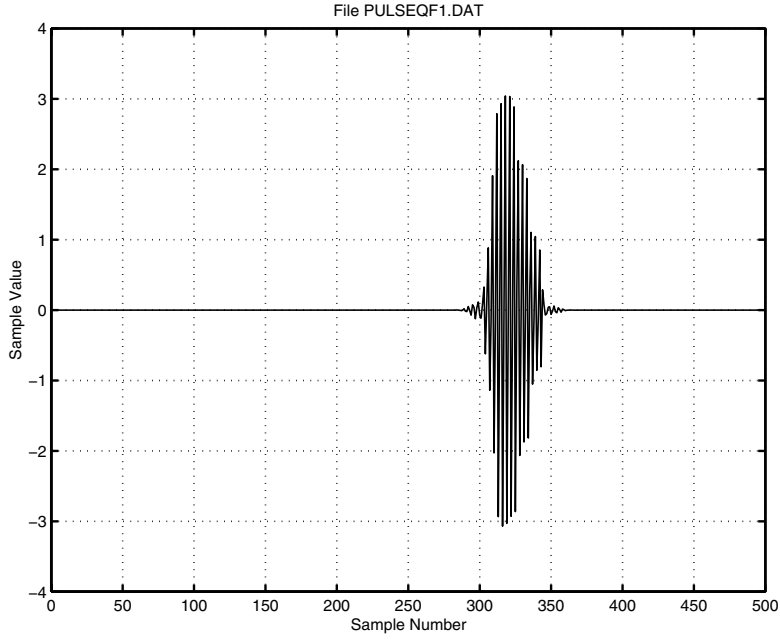
As can be seen in Figure 4.31(a), the quantization noise of the low-frequency cosine wave is highly correlated with the signal and has a large amount of high-frequency content (indicated by the sharp transitions). When the number of bits used to quantize a narrow bandwidth signal is small, the quantization must be considered a nonlinear process that creates harmonics of the signal. In the example shown in Figure 4.31, the higher-frequency harmonics are dominant so that the lowpass filter eliminated more of the quantization noise than expected.

A more realistic example of filtering a quantized signal is shown in Figure 4.32 using the radar pulse data (file PULSE.DAT as shown in Figure 4.24). Figure 4.32(a) shows the 3-bit quantized signal generated using the QUANTIZE program (file PULSEQ.DAT). Figure 4.32(b) shows the 35-tap FIR highpass filtered result (file PULSEQF1.DAT) obtained using the FIRFILT program. The statistics obtained from the STATS program for these two files are as follows:

```
>STATS

Enter input filename: PULSEQ.DAT

File trailer:

RF pulse data sampled at 3 times center frequency.

Quantized signal and error
Signal max = 0.999940, Int max = 3

Record     Average    Variance    Sigma      Min      Max
------     -------    --------    -----      ---      ---
   0       -0.0020    0.2625     0.5124   -3.0000   3.0000
   1        0.0019    0.0049     0.0701   -0.3927   0.4890

>STATS

Enter input filename: PULSEQF1.DAT

File trailer:

RF pulse data sampled at 3 times center frequency.

Quantized signal and error
Signal max = 0.999940, Int max = 3

Highpass filtered using FIRFILT
```



(a)



(b)

FIGURE 4.32 Filtering of 3-bit quantized version of PULSE.DAT. (a) PULSE.DAT (see Figure 4.24) after quantization (maximum value = 3). (b) After 35-point FIR highpass filtering using FIRFILT.

```
Record      Average     Variance     Sigma       Min       Max
------      -------     --------     -----       ---       ---
   0       -0.0000      0.2541      0.5041    3.0687    3.0416
   1        0.0000      0.0030      0.0545    0.3023    0.3795
```

In this example, the quantization noise was reduced by 2.2 dB, but the signal was reduced by 0.1 dB resulting in a signal-to-noise improvement of only 2.1 dB. This is somewhat less than the 3.9 dB signal-to-noise improvement that would be obtained for the FIR highpass filter with a white Gaussian noise input. In this case, much of the quantization noise is in the same band as the signal, so it is more difficult to eliminate.

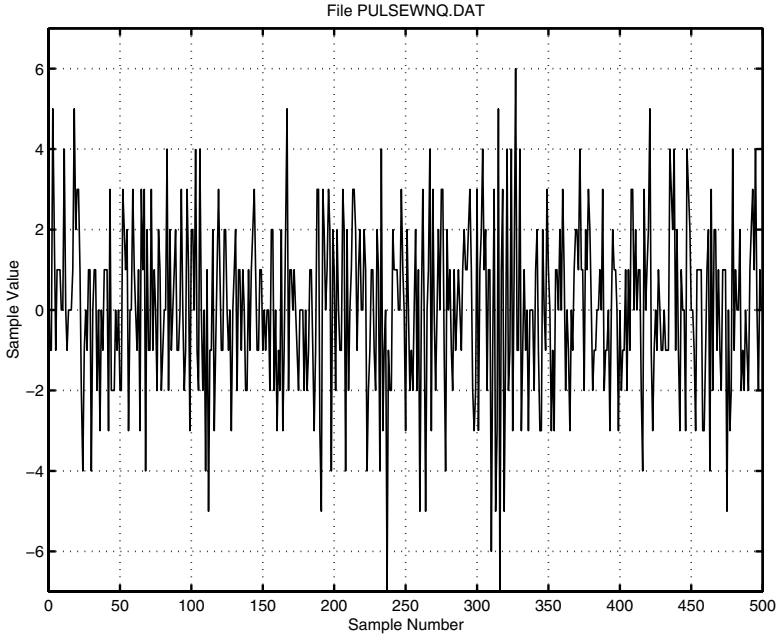
Sometimes quantization noise can be tolerated because the noise level of the input signal is well above the quantization noise level. Such a case is illustrated in Figure 4.33 using the noisy radar pulse data (file PULSEWN.DAT as was used in Section 4.7.3). In this case, a 4-bit quantized version of the noisy radar pulse [shown in Figure 4.33(a)] is filtered using the 52-tap FIR matched filter as was used in Section 4.7.3. The result [shown in Figure 4.33(b)] has a signal-to-noise ratio of 11.6 dB, which is only 0.3 dB less than the unquantized signal-to-noise ratio (11.9 dB) of the result shown in Figure 4.29(b). The quantization noise level was small compared to the white noise level, and the matched filter served to filter both noise sources.

Using the least number of bits in a high-speed signal processing system can be very important because of the cost, complexity, and processing-time reductions associated with processing fewer bits. Thus, whenever the signal-to-noise ratio of a quantized signal is in question, it is important to simulate the quantization process to be sure that the signal quantization at a particular point is not adversely affecting the system performance, increasing the cost, or decreasing the processing efficiency.

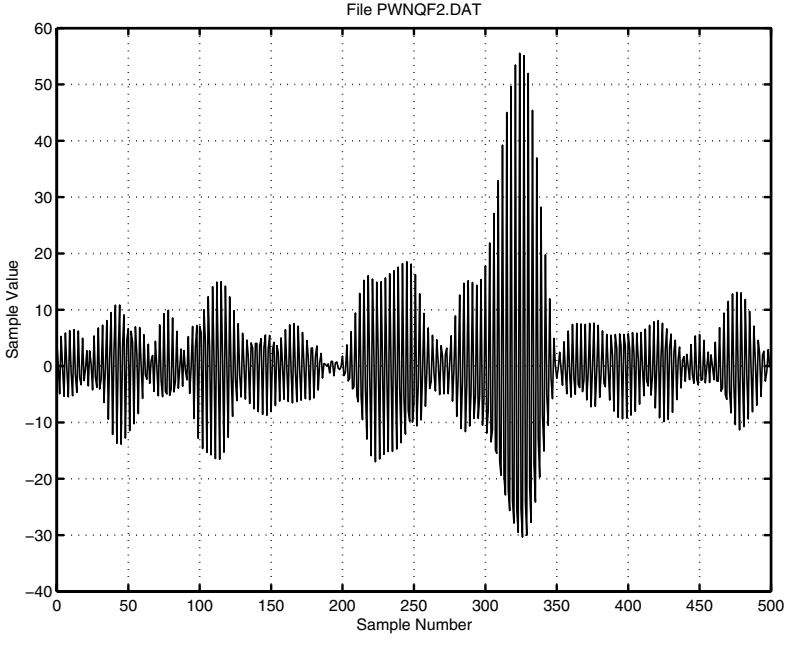
4.8 NONLINEAR FILTERING

Nonlinear filtering covers a vast array of techniques used primarily to process signals that will be analyzed in qualitative ways. For example, in image processing non-linear operations are often performed to increase the contrast or enhance some feature of an image (Chapter 7 covers many of these techniques). Although quantitative measures of the enhancement of the image can be developed, nonlinear filtering is almost always judged based on a qualitative "look" at the result. Nonlinear filtering techniques are not applied nearly as often as linear filtering techniques, perhaps because the nonlinear techniques are less well known and are often developed without a strong theoretical background. Also, the majority of the digital signal processing devices are more efficient when performing FIR or IIR linear filter than nonlinear filtering, which often requires conditional execution based on the input data.

The simplest nonlinear filter is a nonlinear element followed by a linear filter. Examples of this type of nonlinear process include logarithmic compression used in speech signal processing and histogram equalization in image processing. This addition of a nonlinear element makes the analysis of the filtering process much more difficult be-



(a)



(b)

FIGURE 4.33 (a) Plot of 4-bit quantized version (maximum absolute value = 7) of PULEWN.DAT (see Figure 4.28). (b) After 52-point FIR matched filtering using FIRFILT.

cause superposition cannot be applied. In particular, the filtering of noise in a signal is attenuated by different amounts depending on the signal level. The noise level at the output of a nonlinear system can be estimated by passing the signal alone and the signal with noise separately through the system. These two outputs can then be subtracted to determine the output noise level for a particular signal input. A large number of possible signal inputs must be generated and passed through the system in order to estimate the worst case output signal-to-noise ratio. In fact, the quantization filtering examples discussed in Section 4.7.4 are examples of a nonlinear process (the quantization process) followed by a linear filter.

Another class of nonlinear filters involves the replacement of the input data with a new output based on a set of conditions. For example, the output of an FIR lowpass filter could be used to form the output sequence only when the input sequence (delayed to compensate for the filter delay) is different from the filter output by some threshold. This is called *conditional filtering* and is discussed in more detail in Chapter 7, Section 7.3.4.

Perhaps some of the more complicated non-inear filters rearrange the order of the input and generate an output based on this rearrangement. Usually, some number of samples in a windowed section of the input signal are sorted into a descending order and the output sample is generated from the sorted values. If the middle value from the sorted values is taken as the output, the process is called *median filtering.* The median filter output can be used conditionally based on the difference between the median output and the input in which case the process is called *conditional median filtering.* Median filtering techniques are considered in Chapter 1, Section 1.5.2. Section 4.8.2 discusses the implementation of one-dimensional median filtering and Section 7.3.1 of Chapter 7 discusses median filtering of images. The next section describes a simple and efficient way to implement the sorting of an array (required for median filtering) using the **qsort** standard library function.

4.8.1 Sorting

Listing 4.17 shows the function **medianFilter** contained in the file MEDIAN.CPP. The MEDIAN program (see Section 4.8.2) calls this function in order to conditionally median filter the first record of a DSP data file. This section describes the use of the **qsort** standard library function used in the **medianFilter** function to sort the floating-point array passed to it by the caller.

The **qsort** function (see C++ documentation for more details) is an implementation of the recursive quicksort algorithm developed by Hoare in 1962 (see Knuth for a detailed discussion of sorting algorithms). Given an array of values to be sorted, one of the elements is selected and the other elements are partitioned into two subsets based on the value of the selected element. One subset contains elements less than the selected element (placed at the end of the list), and the other subset contains the rest. This partitioning process is then applied recursiveIy to each of the subsets. When a subset has only one element, no more subsets can be created and the recursion stops.

```
//////////////////////////////////////////////////////////////////////////////
//
// int cmp( const void *a, const void *b )
//   Compare two floats for sorting ( qsort )
//
//  Returns:
//   1 -- *a > *b
//   -1 -- *a < *b
//   0 -- *a == *b
//
//////////////////////////////////////////////////////////////////////////////
int cmp( const void *a, const void *b )
{
    if( *(float *)a > *(float *)b )
        return 1;
    else if( *(float *)a < *(float *)b )
        return -1;
    else
        return 0;
}
//////////////////////////////////////////////////////////////////////////////
//
// medianFilter(const Vector<float>& vIn, int lenFilt, float thresh )
//   Conditionally median filters a vector.
//
//   If the absolute difference of the median and the center
//   sample exceed the threshold parameter, then the sample is
//   replaced by the median ( a threshold of zero will give strait
//   median filtering ).
//
//  Throws:
//   DSPException
//
//  Returns:
//   Resultant filtered vector
//
//////////////////////////////////////////////////////////////////////////////
Vector<float> medianFilter( const Vector<float>& vIn, int lenFilt, float thresh )
{
    // Vector to return ( copy input to start with )
    Vector<float> vRet = vIn;

    // Data to sort
    float *arraySort = new float[lenFilt];
    if( arraySort == NULL )
```

LISTING 4.17 Function **medianFilter** used by program MEDIAN to median filter an array of floating-point data. (*Continued*)

```
        throw DSPMemoryException( "Sort array" );

    int mlen2 = lenFilt / 2;

    for( int i = 0; i < ( vIn.length() - lenFilt ); i++ )
    {
        // Copy data to sort
        for( int j = 0; j < lenFilt; j++ )
            arraySort[j] = vIn[i + j];

        // Sort the data using qsort standard library routine
        qsort( arraySort, lenFilt, sizeof( float ), cmp );

        // Replace with median if difference greater than threshold
        if( fabs( arraySort[mlen2] - vIn[i + mlen2] ) > thresh )
            vRet[i + mlen2] = arraySort[mlen2];
    }
    delete [] arraySort;
    return vRet;
}
```

LISTING 4.17 (*Continued*)

The **medianFilter** function extracts a group of floating-point values from the input array (the length of this group of values is equal to the median filter length) and copies the data to the **arraySort**. This data is sorted using **qsort** as follows:

```
qsort( arraySort, lenFilt, sizeof(float), cmp );
```

The **qsort** function is designed to work with any type of data array (even arrays of strings). The first argument (**arraySort**) is a pointer to the array to sort. The second and third arguments give the number of elements in the array and the size of each element. The last argument is a pointer to a function that performs the comparison on a pair of arguments of the same data type as the array.

Pointers to functions have not been discussed previously because they are not used frequently in DSP applications. In C++, a function is not a variable, but it does start at a particular memory location so it is possible to define pointers to functions. Pointers to functions can be assigned, used as a function call [the function pointed to by **ptr** is called using **(*ptr)()** if the function does not require arguments], placed in arrays, passed to functions, and so on. In the case of the **qsort** function, the function pointer is passed to **qsort** so that it can call the comparison function to determine how to order the array of data. The comparison routine used in **medianFilter** is as follows:

```
int float_cmp(a,b)
    float *a,*b;
```

```
{
    if(*a > *b) return (1);
    else return (-1);
}
```

This short function takes two pointers to float variables (which are supplied by the **qsort** function) and compares the floating-point values. If the first is greater than the second, an integer equal to 1 is returned; otherwise, -1 is returned. The sign of the returned value is used by **qsort** to partition the array.

4.8.2 Median Filtering

The **medianFilter** function (see Listing 4.17) determines the median value in a sliding window (the number of points in the window is given by the variable **lenFilt**) and conditionally replaces the input data with the median value. The input floating-point array is first copied to the output floating-point array (both of which are allocated by the calling function). Then, if the difference between the median value and the input is greater than the threshold, the output data is replaced by the median value. Note that the input sequence remains unchanged. If the threshold is set to zero, then the filtering performed is simple median filtering where the median value is always used. When the threshold is some positive value, conditional median filtering is implemented. Listing 4.18 shows the program MEDIAN, which uses the **medianFilter** function to median filter the first record in a DSP data file. The following computer dialogue illustrates the use of the MEDIAN program with the single cosine wave with added Gaussian noise [data file W05WN.DAT as shown in Figure 4.27(b)]:

```
Enter input file name : W05WN.DAT

File trailer:

Signal of length 100 equal to the sum of 1
cosine waves at the following frequencies:
f/fs = 0.050000

Noise signal of length 100, Gaussian distribution, Amplitude = 0.2

Enter median filter length [3...100] : 3

Enter conditional median threshold [0...1e+010] : 0

Enter output file name : W05WNM30.DAT
```

The result of the above use of the MEDIAN program is shown in Figure 4.34(a). Note the flat areas at the extremes of the filtered signal that would result from the median filtering of a cosine wave with or without noise. This “flat topping” is generally unwanted when

```
////////////////////////////////////////////////////////////////////////////
//
// median.cpp - Median filter
//
////////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "image.h"
#include "get.h"
////////////////////////////////////////////////////////////////////////////
//
// int main()
//   Program to conditionally median filter a record and
//   generate a DSPFile containing the new filtered data.
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Open the input file
        do getInput( "Enter input file name", strName );
        while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        Vector<float> vIn;
        dspfile.read( vIn );

        // Check if file is empty
        if( vIn.isEmpty() )
            throw DSPFileException( "File empty" );

        // Get file trailer
        String strTrailer;
        dspfile.getTrailer( strTrailer );
        dspfile.close();

        // Show trailer
```

LISTING 4.18 Program MEDIAN used to median filter a data record in a DSP data file. (*Continued*)

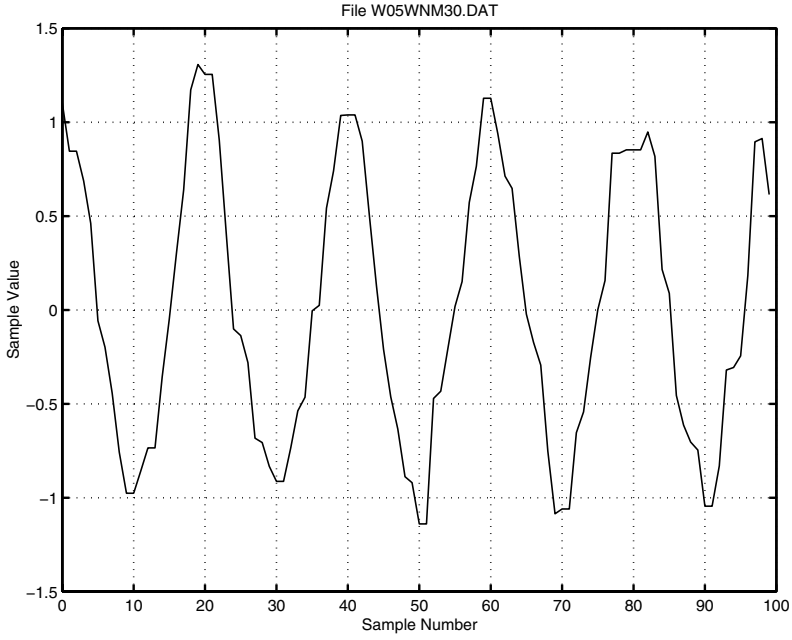
```
        if( strTrailer.isEmpty() == false )
            cout << "File trailer: " << strTrailer << endl;

        // Get user parameters
        int lenFilt = 0;
        getInput("Enter median filter length",lenFilt,3, vIn.length());
        float thresh = 0.0f;
        getInput("Enter conditional median threshold",thresh, 0.0,1.e10);

        Vector<float> vOut = medianFilter( vIn, lenFilt, thresh );

        // Open output DSPFile
        do getInput( "Enter output file name", strName );
        while( strName.isEmpty() );

        dspfile.openWrite( strName );
        dspfile.write( vOut );

        // Make descriptive trailer and write to file
        strTrailer += "Filtered using MEDIAN filter\n";
        dspfile.setTrailer( strTrailer );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 4.18 (*Continued*)

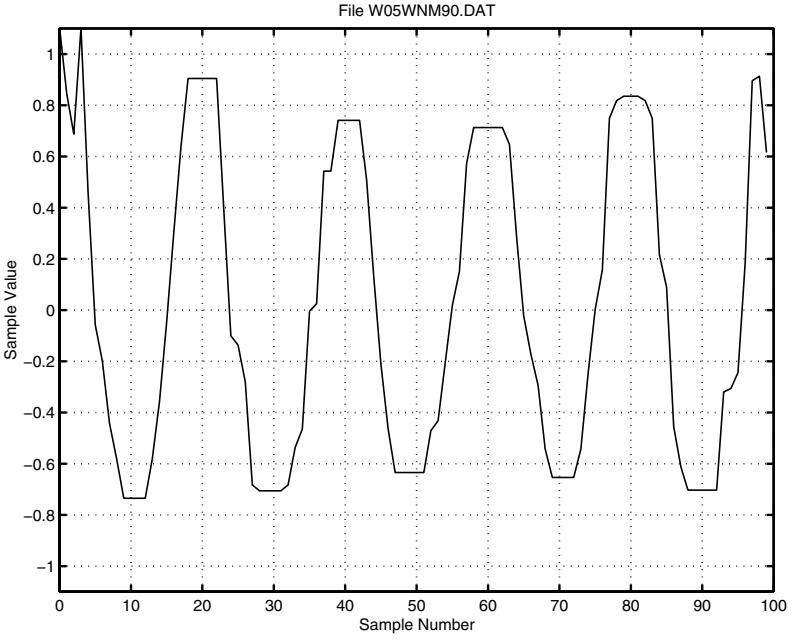
filtering sinusoidal signals but is quite reasonable in a row or column of an image. Figure 4.34(b) shows the result of a 9-point median filter with the same data. The 9-point result has longer flat regions, and will therefore have more lower-frequency components. Figure 4.35 shows a similar result for conditional median filtering with a threshold of 0.1 for the 3-point median filter [Figure 4.35(a)] and a threshold of 0.2 for the 9-point median filter [Figure 4.35(b)]. Note that the degree of "flat topping" is substantially reduced.

4.8.3 Speech Compression

The simplest way to reduce the bandwidth required to transmit speech is to simply reduce the number bits per sample that are sent. If this is done in a linear fashion, then the quality of the speech (in terms of signal-to-noise ratio) will degrade rapidly when less than 8

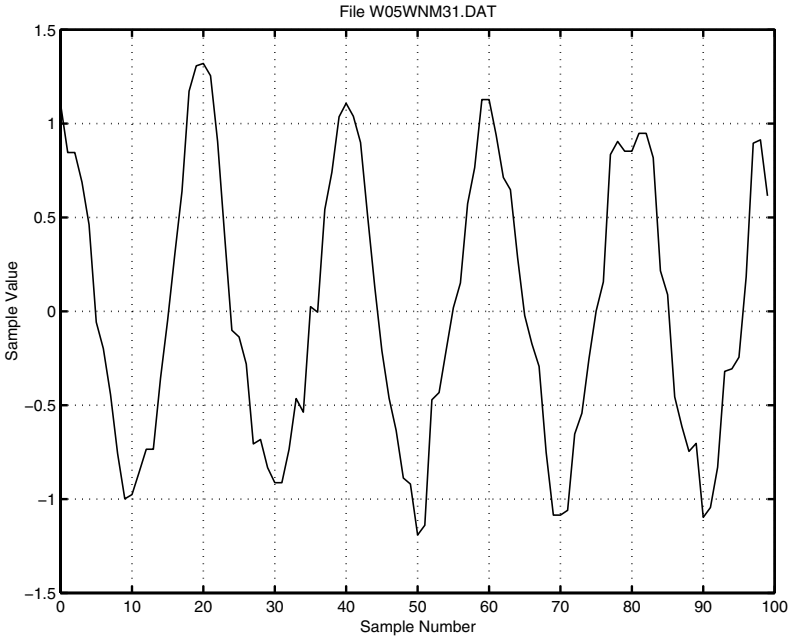


(a)

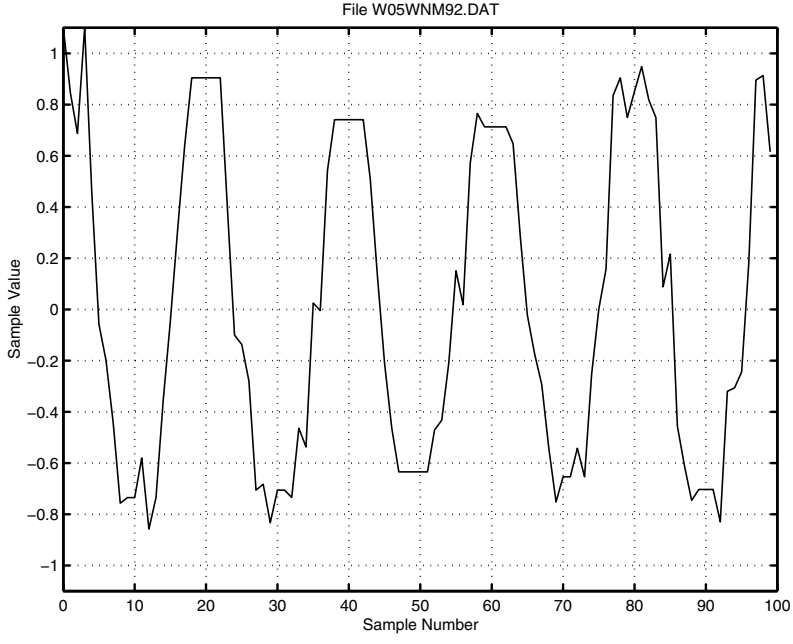


(b)

FIGURE 4.34 Median filtering of W05WN.DAT [as shown in Figure 4.27(b)] using program MEDIAN (threshold = 0). (a) Sort length = 3. (b) Sort length = 9.



(a)



(b)

FIGURE 4.35 Conditional median filtering of W05WN.DAT [as shown in Figure 4.27(b)] using program MEDIAN. (a) Sort length = 3, threshold = 0.1. (b) Sort length = 9, threshold = 0.2.

bits per sample are used. Speech signals require 13 or 14 bits with linear quantization in order to produce a digital representation of the full range of speech signals encountered in telephone applications. The CCITT recommendation G.711 specifies the basic pulse code modulation (PCM) algorithm, which uses a logarithmic compression curve called μ-law. μ-law (see Section 1.5.2 in Chapter 1) is a piecewise linear approximation of a logarithmic transfer curve consisting of 8 linear segments. It compresses a 14-bit linear speech sample down to 8 bits. The sampling rate is 8000 Hz of the coded output. A compression ratio of 1.75:1 is achieved by this method without much computational complexity. Speech quality is not degraded significantly, but music and other audio signals would be degraded. Listing 4.19 shows the program MULAW.CPP, which encodes and decodes a speech signal using μ-law compression. The encode and decode functions that use tables to implement the compression are also shown in this listing. Because the tables are rather long, they are in the include file MU.H.

```
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "mu.h"
//*************************************************************
//
// MULAW.CPP - PROGRAM TO DEMONSTRATE MU LAW SPEECH COMPRESSION
//
//*************************************************************
int main()
{
    try
    {
        DSPFile dspfileIn;
        DSPFile dspfileOut;
        String strName;
        String strTrailer;

        // Open the input file
        do getInput( "Enter input file", strName );
        while( strName.isEmpty() || dspfileIn.isFound( strName ) == false );
        dspfileIn.openRead( strName );
        int lenIn = dspfileIn.getRecLen();
        Vector<float> vIn;
        dspfileIn.read( vIn );
```

LISTING 4.19 Program MULAW.CPP, which encodes and decodes a speech signal using μ-law compression. (*Continued*)

```
        // Create output vector
        Vector<int> vOut( lenIn );

        for(int i = 0 ; i < lenIn ; i++) {
// encode 14 bit linear input to mu-law
            int j = abs(vIn[i]);
            if(j > 0x1fff) j = 0x1fff;
            int k = invmutab[j >> 1];
            if(vIn[i] >= 0) k |= 0x80;
// decode the 8 bit mu-law
            vOut[i] = mutab[k];
        }
        // Write filtered vector out to file
        do getInput( "Enter mu-law output file name", strName );
        while( strName.isEmpty() );
        dspfileOut.openWrite( strName );
        dspfileOut.write( vOut );
        // Close files
        dspfileIn.close();
        dspfileOut.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
```

LISTING 4.19 (*Continued*)

4.9 OSCILLATORS AND WAVEFORM SYNTHESIS

The generation of pure tones is often used to synthesize new sounds in music or for testing DSP systems. The basic oscillator is a special case of an IIR filter where the poles are on the unit circle and the initial conditions are such that the input is an impulse. If the poles are moved outside the unit circle, the oscillator output will grow at an exponential rate. If the poles are placed inside the unit the circle, the output will decay toward zero. The state (or history) of the second-order section determines the amplitude and phase of the future output. The next section describes the details of this type of oscillator. Section 4.9.2 considers another method to generate periodic waveforms of different frequencies—the wave table method. In this case any period waveform can be used to generate a fundamental frequency with many associated harmonics.

4.9.1 IIR Filters as Oscillators

The impulse response of a continuous time second order oscillator is given by

$$y(t) = e^{-dt}\,\frac{\sin(\omega t)}{\omega} \tag{4.18}$$

If d > 0 then the output will decay toward zero and the peak will occur at

$$t_{peak} = \frac{\tan^{-1}(\omega/d)}{\omega} \tag{4.19}$$

The peak value will be

$$y(t_{peak}) = \frac{e^{-dt_{peak}}}{\sqrt{d^2 + \omega^2}} \tag{4.20}$$

A second-order difference can be used to generate an approximation response of this continuous-time output. The equation for a second-order discrete time oscillator is based on an IIR filter and is as follows:

$$y_{n+1} = c_1 y_n - c_2 y_{n-1} + b_1 x_n \tag{4.21}$$

where the x input is only present for t = 0 as an initial condition to start the oscillator and

$$c_1 = 2e^{-d\tau}\cos(\omega\tau)$$
$$c_2 = e^{-2d\tau}$$

where τ is the sampling period ($1/f_s$) and ω is 2π times the oscillator frequency.

The frequency and rate of change of the envelope of the oscillator output can be changed by modifying the values of d and ω on a sample by sample basis. This is illustrated in the OSC program shown in Listing 4.20. The output waveform grows from a peak value of 1.0 to a peak value at the sample number given by **START_SAMPLES**. After this sample is generated the envelope of the output decays toward zero and the frequency is reduced in steps every **CHANGE_SAMPLES** samples. A short example output waveform is shown in Figure 4.36.

4.9.2 Table-Generated Waveforms

Listing 4.21 shows the program WAVETAB.CPP, which generates a fundamental frequency at a particular musical note given by the variable **key**. The frequency in Hertz is relate to the integer **key** as follows:

$$f = 440 \bullet 2^{key/12}$$

```
////////////////////////////////////////////////////////////////////////////
//
// osc.cpp - generate a sine wave with variable envelope
//           using a 2nd order IIR section
//
////////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "dft.h"
#include "disk.h"

////////////////////////////////////////////////////////////////////////////
//
// Constants
//
////////////////////////////////////////////////////////////////////////////
// Length of data to create
const int LENGTH = 800;
// Samples to generate
const int START_SAMPLES = 200;
// Change frequency every so often
const int CHANGE_SAMPLES = 100;
// Starting frequency (Hz)
const float START_FREQ = 1000.0f;

////////////////////////////////////////////////////////////////////////////
//
// float osc( float freq, float rate, int changed )
//    Function to generate samples from a second
//    order oscillator where rate is the envelope rate
//    of change parameter (close to 1). The changed
//    flag indicates that frequency and/or rate have
//    changed.
//
//  Returns:
//    Sample from oscillator
//
////////////////////////////////////////////////////////////////////////////
// Start new sequence from t = 0
const int OSC_START_NEW = -1;
// No change, generate next sample in sequence
const int OSC_NO_CHANGE = 0;
// Change rate or frequency after start
const int OSC_CHANGED = 1;
float osc( float freq, float rate, int changed )
{
```

LISTING 4.20 Program OSC.CPP to generate a sine wave signal with a variable envelope using a second-order IIR section. (*Continued*)

```
    const float TWO_PI_DIV_SAMPLE_RATE = (float)( 2.0f * PI / SAMPLE_RATE );
    static float y1,y0,a,b,arg;
    float out,wosc;

    if( changed != OSC_NO_CHANGE )
    {
        // Assume rate and freq change every time
        wosc = freq * TWO_PI_DIV_SAMPLE_RATE;
        arg = 2.0f * cos( wosc );
        a = arg * rate;
        b = -rate * rate;

        // Re-start case, set state variables
        if( changed == OSC_START_NEW )
        {
            y0 = 0.0f;
            y1 = rate * sin( wosc );
            return y1;
        }
    }

    // Make new sample
    out = a * y1 + b * y0;
    y0 = y1;
    y1 = out;
    return out;
}

int main()
{
    try
    {
        // Start Amplitude of data
        const double amp = 16000.0;
        // Calculate the rate required to get to
        // desired amp in START_SAMPLES samples
        float rate = (float)exp( log( amp ) / START_SAMPLES );

        // Start at START_FREQ
        float freq = START_FREQ;

        // Open oscillation file
        DSPFile dspfile;
        dspfile.openWrite( "osc.dat" );
```

LISTING 4.20 (*Continued*)

```
        // First call to start up oscillator
        dspfile.writeElement(osc( freq, rate, OSC_START_NEW ) );

        // Special case for first samples to increase amplitude
        for( int i = 0; i < START_SAMPLES; i++)
            dspfile.writeElement(osc( freq, rate, OSC_NO_CHANGE ));

        // Decay the osc 10% every START_SAMPLES samples
        rate = (float)exp( log( 0.9f ) / START_SAMPLES );

        while( i < LENGTH )
        {
            // Change freq every CHANGE_SAMPLES samples
            freq = 0.98 * freq;
            dspfile.writeElement(osc( freq, rate, OSC_CHANGED ));
            i++;
            for( int j = 1 ; j < CHANGE_SAMPLES ; j++ )
            {
                dspfile.writeElement(osc( freq, rate, OSC_NO_CHANGE));
                i++;
            }
        }

        // Close file
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 4.20 (*Continued*)

Thus, a **key** value of zero will give 440 Hz, which is the musical note A above middle C. The WAVETAB.CPP program starts at a key value of -24 (two octaves below A) and steps through a chromatic scale to key value 12 (one octave above A). A total of 37 one-second notes are generated by the program. Each sample output value is calculated using a linear interpolation of the 300 values in the table **gwave**. The first 8,000 sample values generated by WAVETAB are shown in Figure 4.37. This is the first note generated by the program at a frequency of 110 Hz (with a sample rate of 11025). The **gwave** array is 301 elements to make the linear interpolation more accurate (aliasing for some

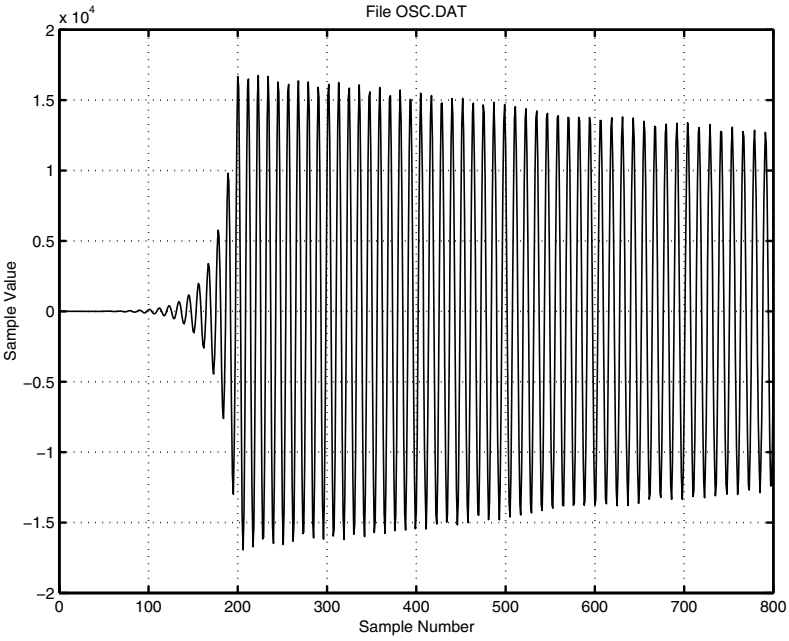


FIGURE 4.36 Example signal output from the OSC.CPP program (modified to reach peak amplitude in 200 samples and change frequency every 100 sample for display purposes).

values of dec could cause a problem if less than 300 elements were used). The first element (0) and the last element (300) are the same, creating a circular interpolated waveform. Any waveform can be substituted to create different sounds. The amplitude of the output is controlled by the **env** variable and grows and decays at a rate determined by **trel** and **amp** arrays.

```
////////////////////////////////////////////////////////////////////////
//
// wavetab.cpp - Wavetable Sound Generator
//
////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"

// gwave[301]
```

LISTING 4.21 Program WAVETAB.CPP to generate periodic waveform at any frequency. (*Continued*)

```
#include "gwave.h"

//////////////////////////////////////////////////////////////////////
//
// Constants
//
//////////////////////////////////////////////////////////////////////
const float TREL[] = {    0.02f,    0.14f,     0.6f, 1.0f, 0.0f };
const float AMPS[] = { 25000.0f, 15000.0f, 6000.0f, 10.0f, 0.0f };
int main()
{
    try
    {
        DSPFile dspfile;

        // Open output file
        dspfile.openWrite( "wavetab.dat" );

        float rates[10] = { 0.0f };
        int tbreaks[10] = { 0 };

        // Dimension of original wave
        int sizeWave = 300;

        // 1 second notes
        int endi = SAMPLE_RATE;

        for( int key = -24 ; key <= 12 ; key++ )
        {
            // Decimation ratio for key semitones down from 440 Hz
            float dec = ((float)sizeWave / SAMPLE_RATE ) *
                440.0f * pow( 2.0f, 0.0833333333f * key );

            // Calculate the rates required to get the desired amps
            int told = 0;
            float ampold = 1.0f;
            // Always starts at unity
            for( int i = 0; i < ELEMENTS( AMPS ); i++ )
            {
                float t = TREL[i] * endi;
                rates[i] = exp(log(AMPS[i] / ampold) / ( t - told));
                ampold = AMPS[i];
                tbreaks[i] = told = t;
            }
            float phase = 0.0f;
```

LISTING 4.21 (*Continued*)

```
            float rate = rates[0];
            float env = 1.0f;
            int ci = 0;
            for( i = 0; i < endi; i++ )
            {
                // Calculate envelope amplitude
                if( i == tbreaks[ci] )
                    rate = rates[++ci];
                env = rate * env;

                // Determine interpolated sample value from table
                int k = (int)phase;
                float frac = phase - (float)k;
                float sample = gwave[k];

                // Possible sizeWave+1 access
                float delta = gwave[k+1] - sample;

                sample += frac * delta;

                // Calculate output and send to DAC
                dspfile.writeElement( env * sample );

                // Calculate next phase value
                phase += dec;
                if( phase >= sizeWave )
                    phase -= sizeWave;
            }
        }

        // Close file
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 4.21 (*Continued*)

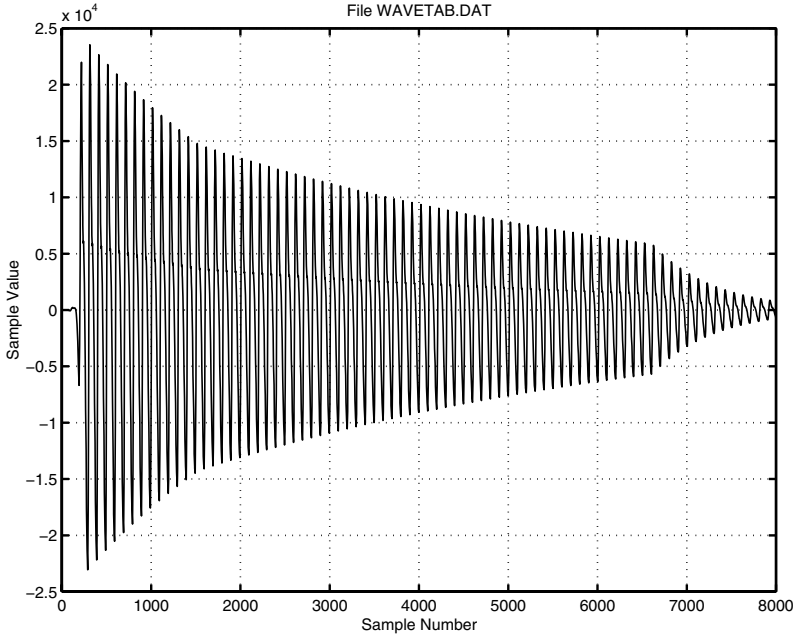


FIGURE 4.37 First 8,000 samples of waveform generated by program WAVETAB.CPP.

4.10 ADAPTIVE FILTERING AND MODELING OF SIGNALS

The parametric approach to spectral estimation attempts to describe a signal as a result from a simple system model with a random process as input. The result of the estimator is a small number of parameters that completely characterize the system model. If the model is a good choice, then the spectrum of the signal model and the spectrum from other spectral estimators should be similar. The most common parametric spectral estimation models are based on AR, MA, or ARMA random process models as was discussed in Section 1.8 of Chapter 1. Three simple applications of these models are presented in the next sections.

A signal can be effectively improved or enhanced using adaptive methods if the signal frequency content is narrow compared to the bandwidth and the frequency content changes with time. If the frequency content does not change with time, a simple matched filter will usually work better with less complexity. The basic LMS (least mean square) algorithm is illustrated in the next section.

4.10.1 LMS Signal Enhancement

Figure 4.38 shows the block diagram of an LMS adaptive signal enhancement that will be used to illustrate the basic LMS algorithm. This algorithm was described in Section 1.7.2 in Chapter 1. The input signal is a sine wave with added white noise. The adaptive LMS

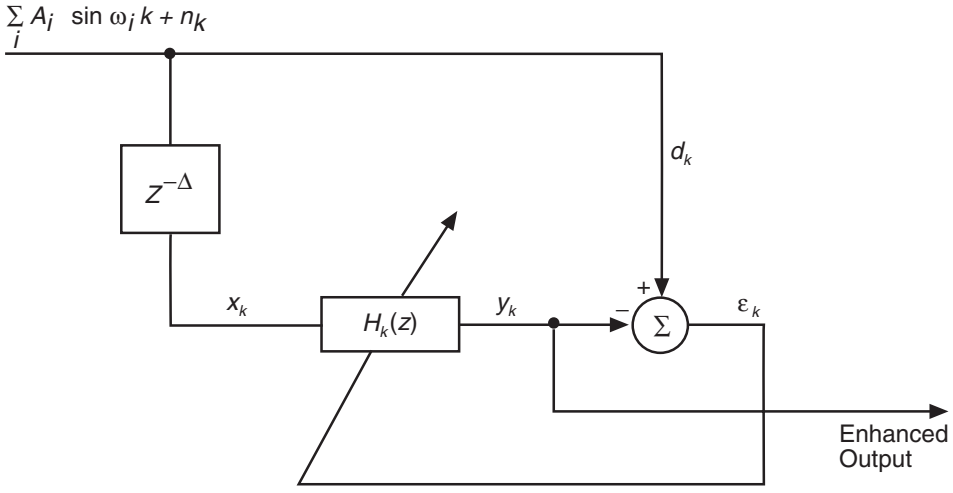


FIGURE 4.38 Block diagram of LMS adaptive signal enhancement.

algorithm (see Listing 4.22 and Listing 4.23) is a 21-tap (20th-order) FIR filter where the filter coefficients are updated with each sample. The desired response in this case is the noisy signal, and the input to the filter is a delayed version of the input signal. The delay is selected such that the noise components of d_k and x_k are uncorrelated (a one-sample delay works well for white noise).

The convergence parameter **mu** is the only input to the program. Although many researchers have attempted to determine the best value for mu, no universal solution has

```
//////////////////////////////////////////////////////////////////////
//
// lms.cpp - LMS Signal Enhancement Demonstration
//
//  Inputs:
//   DSP data filename with signal and noise
//
//  Outputs:
//   DSPFile
//
//////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
```

LISTING 4.22 Program LMS.CPP, which illustrates signal-to-noise enhancement using the LMS algorithm. (*Continued*)

```
// Number of samples
const int SAMPLES = 351;
// Filter order (length = L + 1)
const int FILTER_ORDER = 20;

int main()
{
    try
    {
        // Signal with number of samples
        Vector<float> d( SAMPLES );

        // Filter coefficients
        Vector<float> b( FILTER_ORDER + 1 );
        b = 0.0;

        // Create signal plus noise
        float signalAmp = sqrt( 2.0 );
        float noiseAmp = 0.2 * sqrt( 12.0 );
        float arg = 2.0f * PI / 20.0f;
        for( int k = 0; k < d.length(); k++ )
            d[k] = signalAmp*sin( arg*k ) + noiseAmp * gaussian();

        // Set convergence parameter
        float mu = 0.01f;

        // Scale based on order of filter
        mu = 2.0f * mu / ( FILTER_ORDER + 1 );

        // Open file to write
        DSPFile dspfile;
        dspfile.openWrite( "lms.dat" );

        float x = 0.0f;
        for( k = 0; k < d.length(); k++ )
        {
            float y = lms( x, d[k], b, mu, 0.01F );
            dspfile.writeElement( y );

            // Delay x one sample
            x = d[k];
        }
        dspfile.close();
    }
    catch( DSPException& e )
```

LISTING 4.22 (*Continued*)

```
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 4.22 (*Continued*)

```
////////////////////////////////////////////////////////////////////////////
// lms(float x, float d, const Vector<float>& b, float mu, float alpha )
//   Implements NLMS Algorithm b(k+1)=b(k)+2*mu*e*x(k)/((l+1)*sig)
//
//  Where:
//   x -- Input data
//   d -- Desired signal
//   b -- Adaptive coefficients of FIR filter
//   mu -- Convergence parameter (0.0 to 1.0)
//   alpha -- Forgetting factor:
//      sig(k)=alpha*(x(k)**2)+(1-alpha)*sig(k-1)
//      (>= 0.0 and < 1.0)
//
//  Retuns:
//   Filter output
//
////////////////////////////////////////////////////////////////////////////
// Maximum filter order
const int MAX_FILTER_ORDER = 50;
float lms(float x, float d, Vector<float>& b, float mu, float alpha )
{

    // Start sigma forgetting factor at 2 and update internally
    static float sigma = 2.0f;

    const int filterOrder = b.length() - 1;

    static float px[MAX_FILTER_ORDER + 1] = { 0.0f };
    px[0] = x;

    // Calculate filter output
    float y = b[0] * px[0];
```

LISTING 4.23 Function **lms(x,d,b,l,mu,alpha)** implements the LMS algorithm. (*Continued*)

```
    for( int coefIndex = 1; coefIndex <= filterOrder; coefIndex++ )
        y = y + b[coefIndex] * px[coefIndex];

    // Error signal
    float e = d - y;

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

    // Update coefficients
    for( coefIndex = 0; coefIndex <= filterOrder; coefIndex++ )
        b[coefIndex] = b[coefIndex] + mu_e * px[coefIndex];

    // Update history
    for( coefIndex = filterOrder; coefIndex >= 1; coefIndex— )
        px[coefIndex] = px[coefIndex-1];
    return y;
}
```

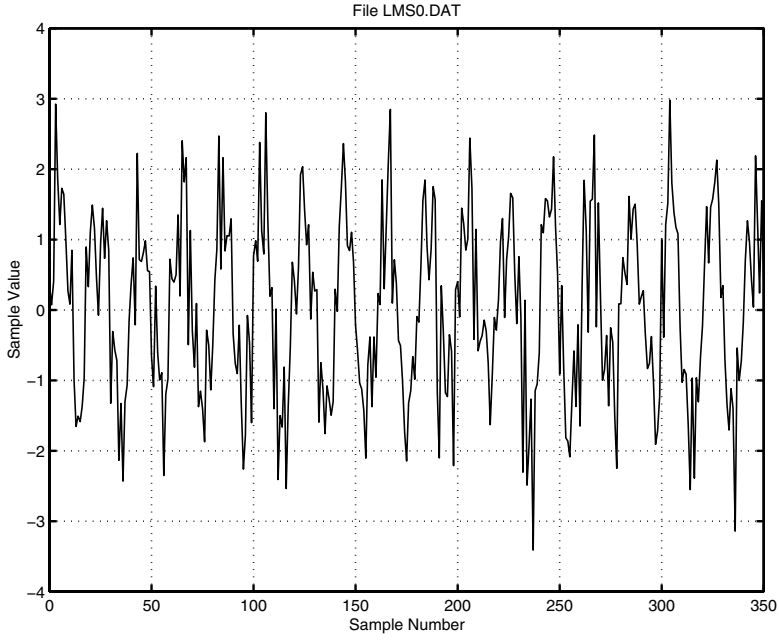
LISTING 4.23 (*Continued*)

been found. If mu is too small, the system may not converge rapidly to a signal as is illustrated in Figure 4.39. The adaptive system is moving from no signal (all coefficients are zero) to an enhanced signal. This takes approximately 300 samples in Figure 4.39(b) with **mu** = 0.01 and approximately 30 samples in Figure 4.39(c) with **mu** = 0.1.

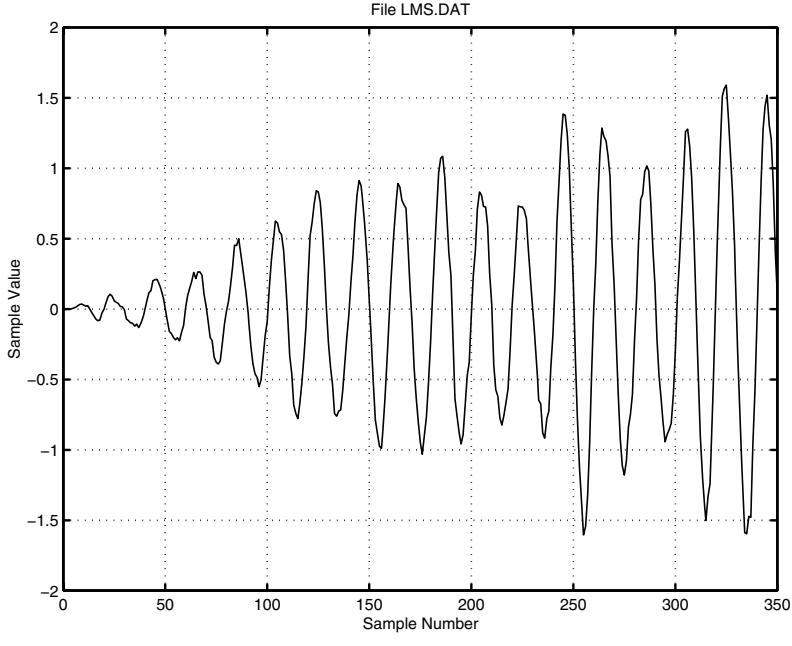
4.10.2 ARMA Modeling of Signals

Figure 4.40 shows the block diagram of a system-modeling problem used to illustrate the adaptive IIR LMS algorithm discussed in detail in Section 1.7.2 in Chapter 1. Listing 4.24 shows the main program ARMA.CPP, which first filters white noise (generated using the Gaussian noise generator described in Section 4.2.1) using a second-order IIR filter and then uses the LMS algorithm to adaptively determine the filter function.

Listing 4.25 shows the function **iirBiquad** which is used to filter the white noise and Listing 4.26 shows the adaptive filter function, which implements the LMS algorithm in a way compatible with real-time input. Although this is a simple ideal example where exact convergence can be obtained, this type of adaptive system can also be used to model more complicated systems such as communication channels or control systems. The white noise generator can be considered a training sequence known to the algorithm, and the algorithm must determine the transfer function of the system. Figure 4.41 shows the error function for the first 7000 samples of the adaptive process. The error reduces relatively slowly due to the poles and zeroes that must be determined. FIR LMS algo-

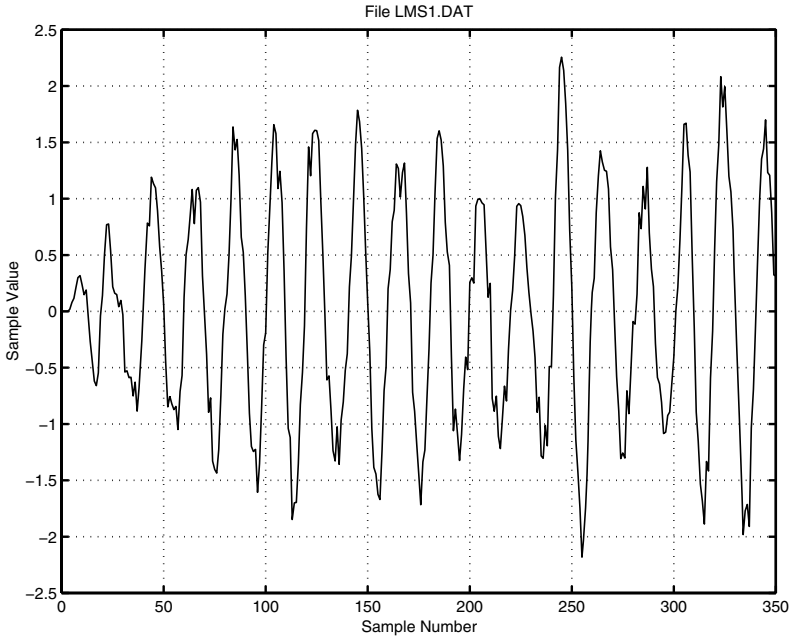


(a)



(b)

FIGURE 4.39 (a) Original noisey signal used in program LMS.CPP. (b) Enhanced signal obtained from program LMS.CPP with **mu** = 0.01. (c) Enhanced signal obtained from program LMS.CPP with **mu** = 0.1.



(c)

FIGURE 4.39 (*Continued*)

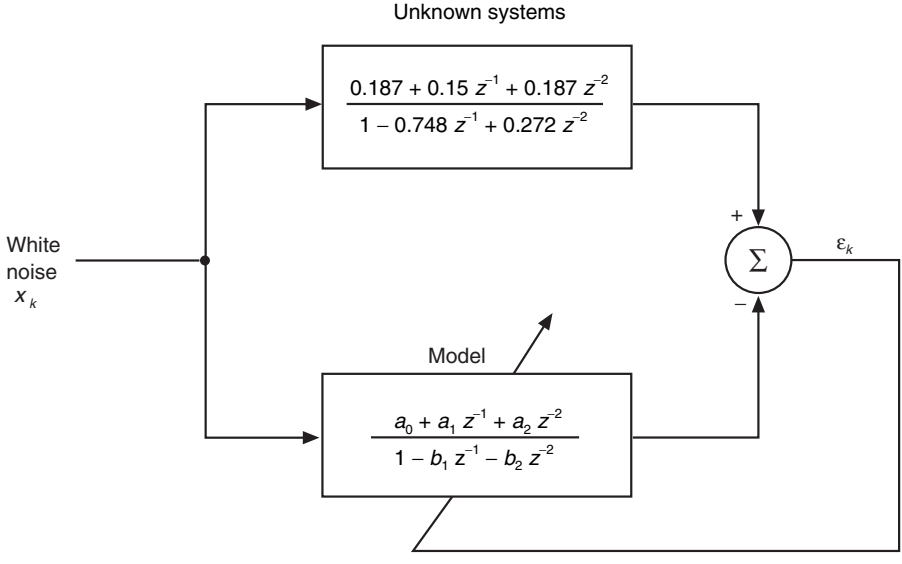


FIGURE 4.40 Block diagram of adaptive system modeling example implemented by program ARMA.CPP.

```
//////////////////////////////////////////////////////////////////
//
// ARMA.CPP - Addaptive IIR filtering
//
//////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"

const int LEN = 7000;

//////////////////////////////////////////////////////////////////
//
// int main()
//
//  Returns:
//   0 -- Success
//
//////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        float a[3] = { 0.187218F, 0.149990698F, 0.187218F };
        float b[2] = { 0.7477891445F, -0.2722149193F };
        Vector<float> d( LEN );

        // Set random seed to known value
        srand( 1 );
        for( int i = 0; i < LEN; i++ )
            d[i] = iir_biquad( gaussian(), a, b );

        // Clear all coefficients at start of adaptive process
        b[0] = b[1] = 0.0f;
        a[0] = a[1] = a[2] = 0.0f;
        Vector<float> y(LEN), a0(LEN), a1(LEN), a2(LEN), b0(LEN), b1(LEN);

        // Reset the random seed to re-generate the random sequence
        srand( 1 );
        for( i = 0; i < LEN; i++ )
        {
            y[i] = iir_adapt_filter( gaussian(), d[i], a, b );
            // print results every 100 adaptations
            if(i%100 == 0)
            {
```

LISTING 4.24 Program ARMA.CPP to demonstrate ARMA modeling of a system. (*Continued*)

```
                cout << d[i] << " " << y[i] << " " << a[0] << " ";
                cout << a[1] << " " << a[2] << " " << b[0] << " ";
                cout << b[1] << endl;
            }
            // save coefficients in vectors
            a0[i] = a[0];
            a1[i] = a[1];
            a2[i] = a[2];
            b0[i] = b[0];
            b1[i] = b[1];
        }
        // Open output file
        DSPFile dspfile;
        String strName;

        do getInput( "Enter output file name", strName );
        while( strName.isEmpty() );
        dspfile.openWrite( strName );
        // write error vector
        dspfile.write( sub(y,d) );
        // write coefficent vectors
        dspfile.write( a0 );
        dspfile.write( a1 );
        dspfile.write( a2 );
        dspfile.write( b0 );
        dspfile.write( b1 );

    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 4.24 (*Continued*)

```
////////////////////////////////////////////////////////////////
//
// float iir_biquad( float input, float *a, float *b )
//   IIR filter single biquad with 2 poles (2 b coefs)
//   and 2 zeros (3 a coefs).
//
//  Returns:
//   Filtered output
//
////////////////////////////////////////////////////////////////
float iir_biquad( float input, float *a, float *b )
{
    if( a == NULL || b == NULL )
        throw DSPParamException( "NULL coefficients" );

    static float out_hist1 = 0.0f;
    static float out_hist2 = 0.0f;
    static float in_hist1 = 0.0f;
    static float in_hist2 = 0.0f;

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

    // Zeros
    output += input * a[0];
    output += in_hist1 * a[1];
    output += in_hist2 * a[2];

    // Update history
    in_hist2 = in_hist1;
    in_hist1 = input;

    out_hist2 = out_hist1;
    out_hist1 = output;

    return output;
}
```

LSTING 4.25 Function **iirBiquad(input,a,b),** which implements one second-order IIR filter (contained in ARMA.CPP).

```
//////////////////////////////////////////////////////////////////////
//
// float iir_adapt_filter(float input, float d, float *a, float *b)
//   Adaptive IIR filter biquad with 2 poles (2 b coefs)
//   and 2 zeros (3 a coefs).
//
//  Returns:
//   Filtered output
//
//////////////////////////////////////////////////////////////////////
float iir_adapt_filter(float input, float d, float *a, float *b)
{
    if( a == NULL || b == NULL )
        throw DSPParamException( "NULL coefficients" );

    static float out_hist1 = 0.0f;
    static float out_hist2 = 0.0f;
    static float beta[2] = { 0.0f };
    static float beta_h1[2] = { 0.0f };
    static float beta_h2[2] = { 0.0f };
    static float alpha[3] = { 0.0f };

    static float alpha_h1[3] = { 0.0f };
    static float alpha_h2[3] = { 0.0f };

    static float in_hist[3] = { 0.0f };

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

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

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

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

    // Error calculation
    float e = d - output;
```

LISTING 4.26 Function **iirAdaptFilter(input,d,a,b),** which implements an LMS adaptive second-order IIR filter (contained in ARMA.CPP). (*Continued*)

```
    // Update coefficients
    a[0] += e * 0.2f * alpha[0];
    a[1] += e * 0.1f * alpha[1];
    a[2] += e * 0.06f * alpha[2];

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

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

    // Update history for beta
    for( i = 0; i < 2; i++ )
    {
        beta_h2[i] = beta_h1[i];
        beta_h1[i] = beta[i];
    }

    // Update input/output history
    out_hist2 = out_hist1;
    out_hist1 = output;

    in_hist[2] = in_hist[1];
    in_hist[1] = input;
    return output;
}
```

LISTING 4.26 (*Continued*)

rithms generally converge much faster when the system can be modeled as an MA system (see Section 5.5.2 for an FIR LMS example). Figure 4.42 shows the path of the pole coefficients (b0,b1) as they adapt to the final result where b0=0.748 and b1=-0.272.

4.10.3 AR Frequency Estimation

The frequency of a signal can be estimated in a variety of ways using spectral analysis methods (see Chapter 5). Another parametric approach is based on modeling the signal as resulting from an AR process with a single complex pole. The angle of the pole resulting from the model is directly related to the mean frequency estimate. This model approach can easily be biased by noise or other signals but provides a highly efficient real-time method to obtain mean frequency information.

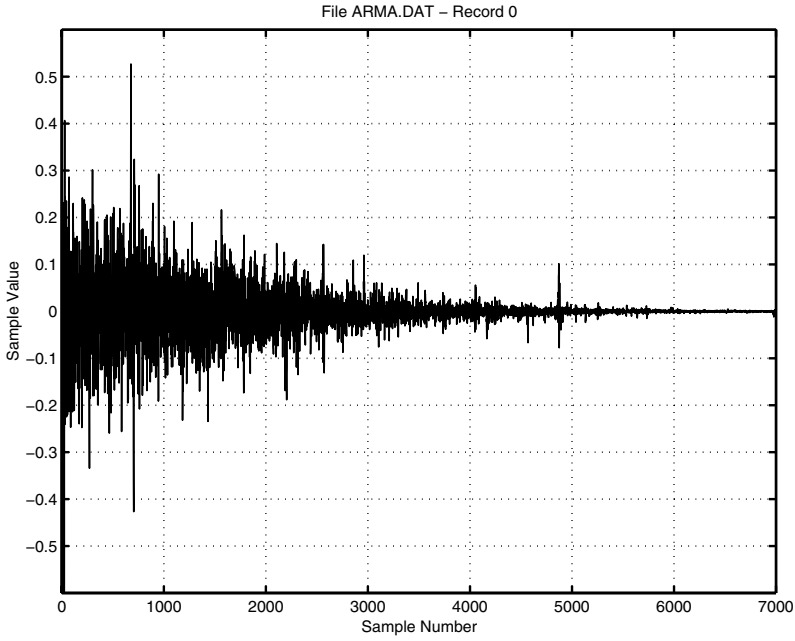


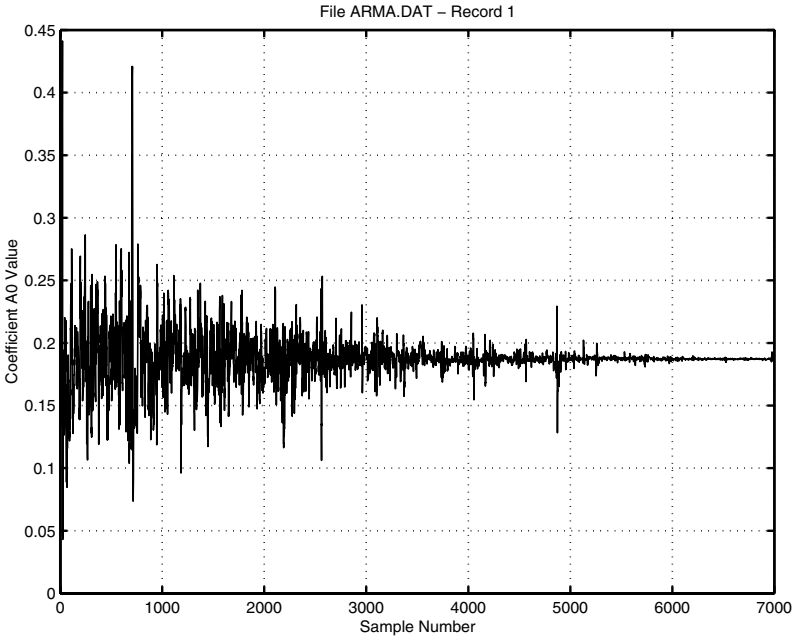
FIGURE 4.41 Error signal during the IIR adaptive process illustrated by the program ARMA.CPP.

The first step in the AR frequency estimation process is to convert the real signal input to a complex signal. This is not required when the signal is already complex as is the case for a radar signal. Real to complex conversion can be done relatively simply by using a Hilbert transform FIR filter. The output of the Hilbert transform filter gives the imaginary part of the complex signal and the input signal is the real part of the complex signal. Listing 4.27 shows the program ARFREQ.CPP, which implements a 35-point Hilbert transform and the AR frequency estimation process. The AR frequency estimate determines the average frequency from the average phase differences between consecutive complex samples. The arc tangent is used to determine the phase angle of the complex results. Because the calculation of the arc tangent is relatively slow, several simplifications can be made so that only one arc tangent is calculated for each frequency estimate. Let x^n be the complex sequence after the Hilbert transform. The phase difference is

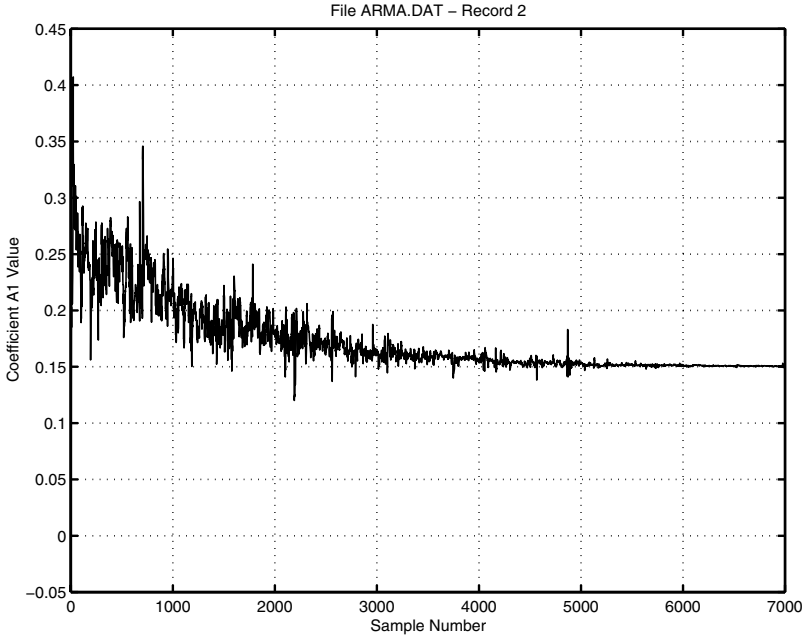
$$\Phi_n = \arg\left[x_n\right] - \arg\left[x_{n-1}\right] = \arg\left[x_n x_{n-1}^*\right]$$

The average frequency estimate is then

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

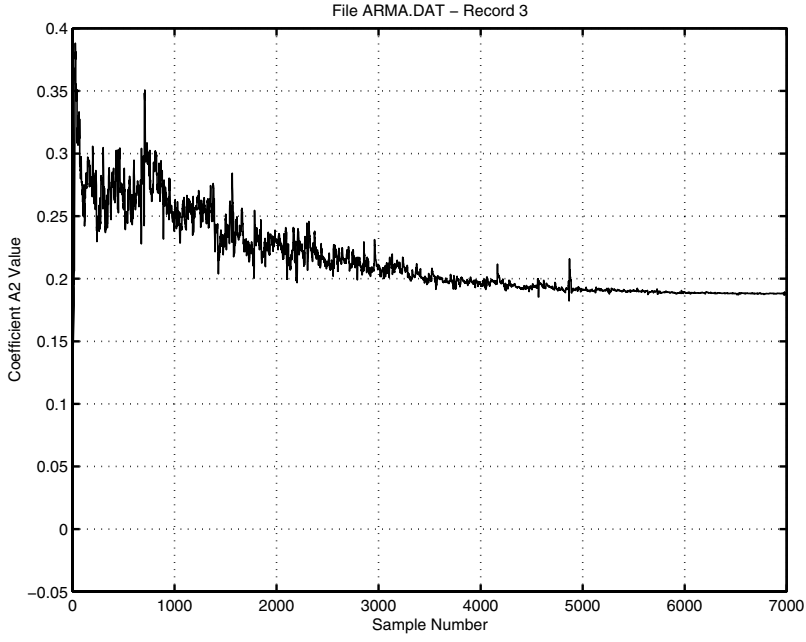


(a)

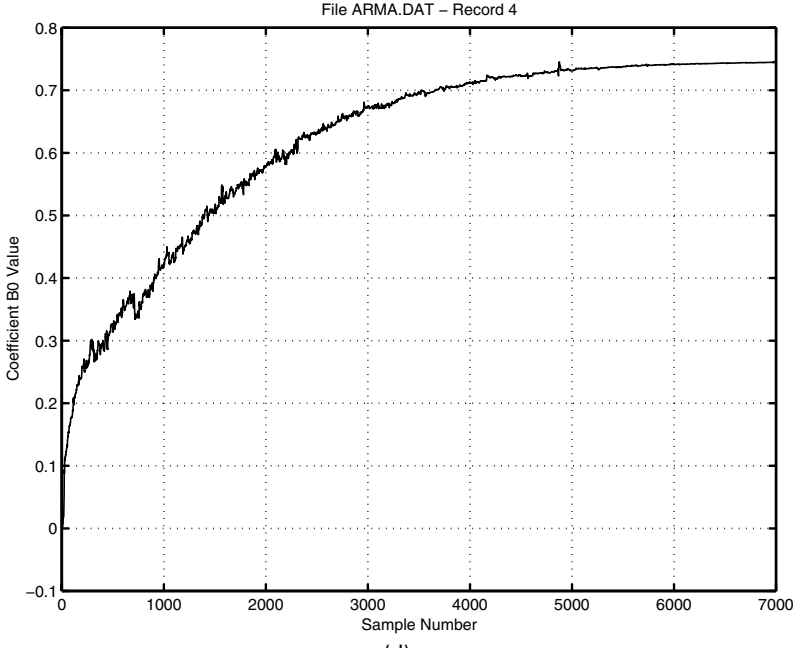


(b)

FIGURE 4.42 Coefficients during the IIR adaptive process illustrated by the program ARMA.CPP. (a) zero coefficient a0, (b) zero coefficient a1, (c) zero coefficient a2, (d) pole coefficient b0, (e) pole coefficient b1. (*Continued*)

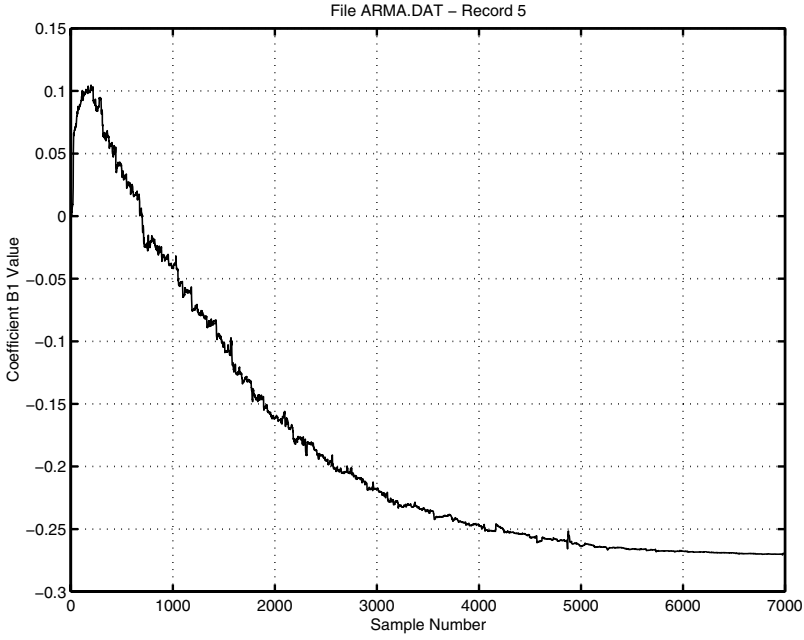


(c)



(d)

FIGURE 4.42 (*Continued*)



(e)

FIGURE 4.42 (*Continued*)

```
////////////////////////////////////////////////////////////////////////////
//
// arfreq.cpp - Calculate first order AR frequency estimate
//
//  Inputs:
//   Filename and length, type, and amplitude of noise
//
//  Outputs:
//   DSPFile with noise added
//
////////////////////////////////////////////////////////////////////////////
```

LISTING 4.27 Program ARFREQ.CPP, which calculates first-order AR frequency estimates. (*Continues*)

```
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "filter.h"
////////////////////////////////////////////////////////////////////////
//
// Constant filter coefficients
//
////////////////////////////////////////////////////////////////////////
// 35 point hilbert transform FIR filter cutoff at 0.02 and 0.48
// +/- 0.5 dB ripple in passband, zeros at 0 and 0.5
float FIRHILBERT35[] =
{
     0.038135F,  0.000000F,  0.024179F,  0.000000F,  0.032403F,
     0.000000F,  0.043301F,  0.000000F,  0.058420F,  0.000000F,
     0.081119F,  0.000000F,  0.120167F,  0.000000F,  0.207859F,
     0.000000F,  0.635163F,  0.000000F, -0.635163F,  0.000000F,
    -0.207859F,  0.000000F, -0.120167F,  0.000000F, -0.081119F,
     0.000000F, -0.058420F,  0.000000F, -0.043301F,  0.000000F,
    -0.032403F,  0.000000F, -0.024179F,  0.000000F, -0.038135F
};
FIRFilter<float> FIRHILBERT( FIRHILBERT35, ELEMENTS( FIRHILBERT35 ) );
////////////////////////////////////////////////////////////////////////
//
// int main()
//   Take real data in one record and determine the 1st
//   order AR frequency estimate versus time. Uses a
//   Hilbert transform to convert the real signal
//   to complex representation
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfileIn;
        DSPFile dspfileOut;
        const float cpi = -1.0f / ( 2.0f * PI );
        float freq;
        int winLen = 32;
        String strName;
```

LISTING 4.27 (*Continues*)

```
        // Open the input file
        do getInput("Enter input file", strName );
        while( strName.isEmpty() || dspfileIn.isFound( strName ) == false );
        dspfileIn.openRead( strName );
        int lenIn = dspfileIn.getRecLen();

        // Open the output file
        do getInput("Enter frequency estimate output file name", strName );
        while( strName.isEmpty() );
        dspfileOut.openWrite( strName );

        Complex sig,last = 0.0f;
        const int delayIndex = FIRHILBERT.length() / 2 - 1;

        for(int InCount = 0 ; InCount < lenIn ; InCount += winLen)
        {

            // Determine the phase difference between samples
            Complex xsum = 0.0f;
            for( int i = 0; i < winLen; i++ )
            {
                float input;
                dspfileIn.readElement( input );
                sig.m_imag = FIRHILBERT.filterElement( input );
                sig.m_real = FIRHILBERT.getHistory( delayIndex );
                xsum = xsum + sig * last;
                // complex conjugate for last
                last.m_real = sig.m_real;
                last.m_imag = -sig.m_imag;
            }

            // Make sure the result is valid, give 0 if not
            if( fabs( xsum.m_real ) > 1e-10f )
                freq = cpi * atan2( xsum.m_imag, xsum.m_real );
            else
                freq = 0.0f;

            dspfileOut.writeElement( freq );
        }
        // Close files
        dspfileOut.close();
        dspfileIn.close();
    }
    catch( DSPException& e )
```

LISTING 4.27 (*Continues*)

```
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 4.27 (*Continues*)

where the last approximation weights the phase differences based on the amplitude of the complex signal and reduces the number of arc tangents to one per estimate. The constant *wlen* is the window length (**winlen** in program ARFREQ) and controls the number of phase estimates that are averaged together. Figure 4.43 shows the results from the ARFREQ program when the CHKL.TXT speech data is used as input. Note that the higher-frequency content of the "chi" sound is easy to identify.

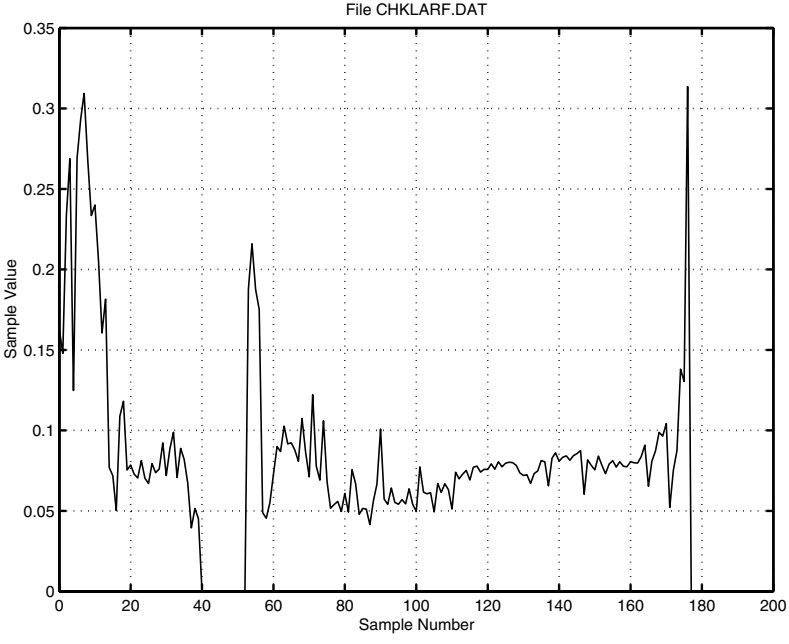


FIGURE 4.43 Frequency estimates from program ARFREQ.CPP using the CHKL.DAT speech data as input.

4.11 EXERCISES

1. Run FIRFILT with CHKL.DAT as the input file, and pick the lowpass filter. Write the result to FIR1.DAT, and then look at it with the WINPLOT program. Run IIRFILT with CHKL.DAT as the input file, and pick the lowpass filter. Write the result to IIR1.DAT, and then look at it with the WINPLOT program. Observe that both IIR and FIR filters give similar results when looking at a speech waveform. Listen to the two different outputs. Can you hear the difference? Also observe that the waveform changes when a highpass filter is used instead of a lowpass filter.
2. Try Exercise 1 using the waveform generated using MKWAVE.CPP in Section 4.2. You should be able to observe differences between the FIR and IIR outputs. Which type of filter is better for this input data?
3. Use the MKWAVE program to generate a cosine wave at a frequency of 0.3072. Run the IIRFILT program with the lowpass filter, and compare this to the result using the FIRFILT lowpass filter. Which type of filter is better at this frequency?
4. Run IIRDEZN program for a one-section second-order butterworth filter with *s*-domain coefficients as follows: sampling rate = 1000, cutoff frequency = 100, a0 = 1, a1 = 0, a2 = 0, b0 = 1, b1 = 1.4142136, b2 = 1. Output to a file (e.g., IIRD1.DAT), and plot using WINPLOT program. This will show the magnitude response on a linear and dB scale (as two different plots in two different consecutive records). What kind of filter is this?
5. Now change the coefficients of Exercise 4 to a0 = 0, a1 = 0, a2 = 1, b0 = 1, b1 = 1.4142136, b2 = 1. What kind of filter is this? How have the digital filter coefficients changed?
6. Now change the coefficients of Exercise 4 to a0 = 0, a1 = 1, a2 = 0, b0 = 1, b1 = 1.4142136, b2 = 1. What kind of filter is this? How have the digital filter coefficients changed?
7. Run the REMEZ program for optimal FIR filter design. Use example 1 for 24-tap lowpass filter. Output to a file (e.g., FIRD1.DAT), and plot using WINPLOT program. This will show the magnitude response on a linear and dB scale (as two different plots in two different consecutive records). Try the other example REMEZ filters, and then try an example of your own by using selection 5 in the REMEZ program. How manny FIR filter taps are required to make a filter with a similar frequency response to the filters used in Exercises 4, 5, and 6?
8. Run IIRDEZN program for second-order bandpass filter *s*-domain coefficients as follows: sampling rate = 8000, cutoff frequency = 1500, a0 = 0, a1 = 0.1, a2 = 0, b0 = 1, b1 = 0.1, b2 = 1. What is the approximate bandwidth and center frequency of this filter? Run FILTALL program to filter the speech data (file CHKL.DAT) using this IIR filter. Listen to the result. Now try this exercise again with a1 = b1 = 0.01. What good is this type of filter?
9. Plot the file PULSE.DAT to observe the radar pulse with no noise. Run ADDNOISE to add white Gaussian noise to the radar pulse. Use a noise multiplier of 0.5, for example. Plot the output of ADDNOISE to observe the RADAR pulse with noise. Suppose you choose a detection threshold of 1.0. How many false detections would there be? Run FIRFILT using the matched filter (selection 2), and observe the result using the WINPLOT program. How many false detections are there now? How much noise can be added before more false detections occur? Design your own FIR filter as was done in Exercise 7. Can you do as well as the matched filter? Hint: The center frequency of the radar pulse is 0.333.
10. Run the REALCMX program to convert the PULSE.DAT pulse data to a baseband complex signal. Use a decimation ratio of 3. Use the WINPLOT program to observe the results. The first output record is the real part of the signal, and the second record is the imaginary part of

the signal. Now try a decimation ratio of 2, and observe the results using WINPLOT. How is the pulse frequency changed by decimation?

11. Run the PSHIFT program using an audio source (music, for example) to change the sample rate of the audio data. Modify the stopband attenuation (**att** in Listing 4.10) and interpolation ratio (**ratio** in Listing 4.10) used in PSHIFT and recompile the program. Then listen to the results. Can you hear the difference? At what point do the reults begin to sound the same?

4.12 REFERENCES

ANTONIOU, A., *Digital Filters: Analysis and Design,* McGraw-Hill, New York, 1979.

BRIGHAM, E. O., *The Fast Fourier Transform and Its Applications*, Prentice Hall, Englewood Cliffs, NJ, 1988.

CROCHIERE, R. E., and RABINER, L. R., *Multirate Digital Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1983.

ELLIOTT, D. F., ed., *Handbook of Digital Signal Processing,* Academic Press, San Diego, 1987.

GHAUSI, M. S., and LAKER, K. R., *Modern Filter Design: Active RC and Switched Capacitor,* Prentice Hall, Englewood Cliffs, NJ, 1981.

IEEE Digital Signal Processing Committee, ed., *Programs for Digital Signal Processing,* IEEE Press, New York, 1979.

JOHNSON, D. E., JOHNSON, J. R., and MOORE, H. P., *A Handbook of Active Filters,* Prentice Hall, Englewood Cliffs, NJ, 1980.

JONG, M. T., *Methods of Discrete Signal and System Analysis,* McGraw-Hill, New York, 1982.

KAISER, J. F., and SCHAFER, R. W., "On the Use of the I_0-Sinh Window for Spectrum Analysis," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-28, no. 1, pp. 105–107, February 1980.

KNUTH, D. E., *Seminumerical Algorithms,* 2nd ed., vol. 2: *The Art of Computer Programming,* Addison-Wesley, Reading, MA, 1981.

MCCLELLAN, J. H., PARKS, T. W., and RABINER, L. R., "A Computer Program for Designing Optimum FIR Linear Phase Digital Filters," *IEEE Transactions on Audio and Electro-Acoustics,* AU-21, no. 6, pp. 506–526, 1973.

MOLER, C., LITTLE, J., and BANGERT, S., *PC-MATLAB User's Guide,* Math Works, Sherborn, MA, 1987.

MOSCHYTZ, G. S., and HORN, P., *Active Filter Design Handbook,* John Wiley & Sons, Inc., New York, 1981.

OPPENHEIM, A., and SCHAFER, R., *Digital Signal Processing,* Prentice Hall, Englewood Cliffs, NJ, 1975.

OPPENHEIM, A., and SCHAFER, R., *Discrete-Time Signal Processing,* Prentice Hall, Englewood Cliffs, NJ, 1989.

PAPOULIS, A., *Probability, Random Variables, and Stochastic Processes,* 2nd ed., McGraw-Hill, New York, 1984.

PARK, S. K. and MILLER, K. W., "Random Number Generators: Good Ones Are Hard to Find," *Communications of the ACM,* October 1988, vol. 31, No. 10.

PARKS, T. W., and BURRUS, C. S., *Digital Filter Design,* John Wiley & Sons, New York, 1987.

PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A., and VETTERLING, W. T., *Numerical Recipes,* Cambridge Press, New York, 1987.

RABINER, L., and GOLD, B., *Theory and Application of Digital Signal Processing,* Prentice Hall, Englewood Cliffs, NJ, 1975.

STEARNS, S. D., and DAVID, R. A., *Signal Processing Algorithms,* Prentice Hall, Englewood Cliffs, NJ, 1988.

VAN VALKENBURG, M. E., *Analog Filter Design*, Holt, Rinehart & Winston, New York, 1982.

ZVEREV, A. I., *Handbook of Filter Synthesis,* John Wiley & Sons, New York, 1967.

CHAPTER 5

Discrete Fourier Transform Routines

Spectrum analysis is the process of determining the correct frequency domain representation of a sequence. From spectrum analysis, information about the underlying frequency content of a sampled waveform such as the bandwidth and central frequency is derived. There are many methods to determine frequency content and the choice of method depends on the characteristics of the signal to be analyzed. Some of the characteristics important in determining the spectrum analysis method are

1. signal-to-noise ratio of the signal
2. statistical character of the noise (Gaussian or other)
3. spectral character of the noise (white, colored)
4. length of sequence compared to rate of sampling
5. any corruption of the signal due to interference

There are two broad classes of spectrum analysis techniques: those based on the Fourier transform and those that are not. By far the most common class of techniques is based on the Fourier transform, and that is the class discussed in this chapter. These techniques begin with the discrete Fourier transform and its inverse. A function that implements the DFT is presented in Section 5.1, and a function for the inverse DFT is presented in Section 5.2.

The reason Fourier transform techniques are so popular and practical is the existence of the fast calculation algorithm, the FFT, that was discussed in detail in Chapter 1. Section 5.3 shows two ways to compute FFTs of radix 2 and gives the code for the Cooley-Tukey algorithm. The inverse FFT is shown in Section 5.4.

Before the DFT is performed on a sequence, a set of window coefficients is often applied to reduce the effects of discontinuity at the waveform boundaries. Functions implementing several popular windows are given in Section 5.5. Section 5.6 introduces a method used to display spectral information, the logarithmic periodogram of the Fourier transform.

Section 5.7 shows how to optimize the FFT for use with real sequences and presents a routine that performs the special computations necessary. The next section, 5.8, puts all the concepts and routines of the chapter together in several examples of spectrum analysis on sampled waveforms. The time-domain waveform, the windowed sequence, and the results of the Fourier transform and inverse Fourier transform are presented. This section shows exactly how to use the spectrum analysis functions in working C code: the parameters required, the data types needed, etc.

Sections 5.9, 5.10, and 5.11 are also applications oriented. They show three special applications of the FFT: fast convolution, power spectrum estimation, and interpolation using the FFT.

5.1 THE DISCRETE FOURIER TRANSFORM ROUTINE

The formula for the DFT as given in Chapter 1 is

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}$$

where the DFT coefficients used in the DFT kernel, *W,* are

$$W_N = e^{-j2\pi/N}$$

The factor $(W_N)^{nk}$ in the DFT equations can be thought of as a function, $W_N(nk)$, with argument *nk,* which can take on any integer value up to $(N-1)^2$. If this function were plotted it would clearly be periodic and repeat every N values of its argument. For purposes of computation then, W_N can be an array of N values whose index is computed as *nk* module *N* (the remainder after *nk* is divided by *N*). Rewriting the DFT equation using the array $W_N[\ ]$ gives the following equation for the DFT:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N[(n\,k) \bmod N]$$

This equation is the basis of the C++ routines for the DFT and inverse DFT called **dft** and **idft.** The C++ language equivalent of the summation is a **for** loop. The only complication with the C++ implementation is the complex arithmetic. This is handled using a **COMPLEX** data structure that consists of two floating-point numbers, referenced as **real** and **imag.** The declaration for the structure **COMPLEX** is as follows:

```
struct {
       float real, imag;
} COMPLEX;
```

This structure is defined in the header file DFT.H for use with the functions in this chapter. A complex array (called **w**) of 16 complex values can then be declared as follows:

```
COMPLEX w[16];
```

Any particular value in the array (with legal values of **i** going from 0 to 15) is accessed using the following code:

```
x = w[i].real;
y = w[i].imag;
```

This code assigns **x** to the real part and **y** to the imaginary part, respectively, of the **i**th entry in the array **w.** Another syntax (which is exactly equivalent to the above form) is

```
x = (w+i)->real
y = (w+i)->imag
```

This method makes more explicit the pointer nature of the value **w,** since the number that is currently assigned to **w** is added to the index **i*sizeof(COMPLEX)** and used to point to the real or imaginary element of the structure. The declaration of **w** as a 16-element COMPLEX array works well as long as we always do a 16-point DFT. If less than 16 points are desired, then we have wasted space. If longer DFTs are required, the program must be compiled again with a larger number than 16 in the declaration. A better solution to this common problem is to use the dynamic memory allocation functions. When this method is used, a **COMPLEX** pointer is declared using

```
COMPLEX *w = new COMPLEX[n];
```

A value is assigned to **w** after sufficient space in memory is set aside for the **COMPLEX** array by the use of the memory allocation function **new**. This statement is packed with information for the compiler. It makes clear the following items:

1. A pointer to a structure of type **COMPLEX** is being assigned a value.
2. This value must have associated with it **n** sections of memory.
3. Each section of memory will consist of the proper number of bytes for a structure of type **COMPLEX.**

After the assignment of a value to the pointer **w** and allocation of sufficient memory for the array the program is ready to use the variable in complex arithmetic. First, a C code

example without pointers—the addition of two complex numbers, **a** and **b** to produce a complex sum **c.**

```
COMPLEX a, b, c;

c.real = a.real + b.real;
c.imag = a.imag + b.imag;
```

First the variables **a, b,** and **c** are declared to be **COMPLEX** structures (not pointers to **COMPLEX** structures). Then the real and imaginary parts are added separately to produce the complex result. Fortunately in C++ the **+** operator can be overloaded, and the same result can be obtained in C++ using: the following code

```
c = a + b;
```

In a similar fashion, the multiply complex operators is also overlaoded as follows:

```
////////////////////////////////////////////////////////////
//
// Complex operator *( const Complex& c1, const Complex& c2 )
//   Multiplies two complex numbers.
//
//  Returns:
//   Complex result.
//
////////////////////////////////////////////////////////////
inline Complex operator *( const Complex& c1, const Complex& c2 )
{
    double a = 0.0;
    a = c1.m_real * c2.m_real;
    a -= c1.m_imag * c2.m_imag;

    double b = 0.0;
    b = c1.m_real * c2.m_imag;
    b += c1.m_imag * c2.m_real;

    Complex ret( a, b );
    return ret;
}
```

With the ability to perform complex vector arithmetic and a set complex values for the exponential factors, we are now ready to write the basic loop for the DFT as follows:

```
COMPLEX *DataIn = new COMPLEX[n];
COMPLEX *DataOut = new COMPLEX[n];
COMPLEX *cf = new COMPLEX[n];
COMPLEX *DinPtr, *DoutPtr, *cfptr;
```

```
    DinPtr = DataIn;
    DoutPtr = DataOut;
    for (int n = 0 ; n < N ; n++){
              p = (long) n*k % N;
              cfptr = cf + p;
              *DoutPtr += *DinPtr * *cfptr;
              DinPtr++;
    }
```

This is similar to the vector multiplication loop, but there are a few important differences. First of all, the index for the **Datain** array and the coefficient array, **cfptr,** are not the same. As mentioned earlier, the exponential factors use **n*k** modulo **N** as their index. The (**long**) operator in the modulus statement is required for long DFTs where the product **n*k** might exceed 16 bits (if an **int** is 16 bits). Also, only one output value is produced by this loop: **DoutPtr** (where **DoutPtr** is a pointer to the **COMPLEX** output array). These values are sums of products, and the **+=** operator is used to produce them. Since what is needed for a complete DFT is all values of **Dataout,** the loop shown above must be nested inside a loop that runs over the index **k** (incrementing the pointer Dataout with each **k** value). With the addition of this outer loop, the DFT function is essentially complete. The complete **dft** function is shown in Listing 5.1.

Note that the DFT calculation is performed from the **Datain** array with the result in the **Dataout** array. Thus, the **dft** function is not an in-place function. The exponential coefficients are stored using a static pointer (**cf**) and are allocated on the first call to **dft.** If future calls to the **dft** function use the same size DFT (as often is done), the coefficients

```
//////////////////////////////////////////////////////////////////////////
//
// Vector<Complex> dft( const Vector<Complex>& vIn )
//   Performs straight DFT on Complex vector vIn.
//
//  Note:
//   vIn does not have to be size power of 2.
//
//  Throws:
//   DSPException
//
//  Returns:
//   Vector<Complex> of transformed vector.
//
//////////////////////////////////////////////////////////////////////////
Vector<Complex> dft( const Vector<Complex>& vIn )
{
```

LISTING 5.1 Function **dft** used to calculate the DFT of a complex array of any length. (*Continued*)

```
// Used to store the coefficients in the complex vector
static Vector<Complex> cf;
// Stores n for future reference
static int nstore = 0;

// Check if empty vIn
if( vIn.isEmpty() )
{
    cf.empty();
    nstore = 0;
    return vIn;
}

int length = vIn.length();

// Create return vector
Vector<Complex> vOut = vIn;

// Check if length changed from last time
if( length != nstore )
{
    // Clear previous coefficients and calculate new values
    cf.empty();
    nstore = length;

    // Create coefficients
    cf.setLength( length );

    double arg = 8.0 * atan( 1.0 ) / length;
    for( int i = 0; i < length; i++ )
    {
        cf[i].m_real = (float)cos( arg * i);
        cf[i].m_imag = -(float)sin( arg * i);
    }
}

// Perform the DFT calculation
for( int k = 0; k < length; k++ )
{
    const Complex *dataIn = vIn.getData( length );
    vOut[k] = *dataIn++;
    for( int n = 1; n < length; n++ )
    {
        int p = ( ( n * k ) % length );

        // Coefficient modulo length
```

LISTING 5.1 (*Continued*)

```
            Complex cfm = cf[p];

            vOut[k] = vOut[k] + (*dataIn * cfm );

            // Go to next input sample
            dataIn++;
        }
    }
    return vOut;
}
```

LISTING 5.1 (*Continued*)

are not calculated again. Whenever the **dft** size changes, however, the coefficients must be recalculated.

5.2 THE INVERSE DISCRETE FOURIER TRANSFORM

The inverse DFT is very similar in form to the forward DFT. The differences are that the coefficients are the complex conjugates of the DFT coefficients and the output must be scaled down by the length of the sequence (this is easily accomplished by scaling the inverse DFT coefficients by $1/N$). The code for the **idft** function that implements the inverse DFT is shown in Listing 5.2. As was done in the **dft** function, the inverse DFT coefficients are stored in a static array to be used by future calls to the **idft** function.

```
//////////////////////////////////////////////////////////////////////
//
// Vector<Complex> idft( const Vector<Complex>& vIn )
//   Performs straight IDFT on Complex vector vIn.
//
//  Note:
//   vIn does not have to be size power of 2.
//
//  Throws:
//   DSPException
//
//  Returns:
//   Vector<Complex> of transformed vector.
//
//////////////////////////////////////////////////////////////////////
Vector<Complex> idft( const Vector<Complex>& vIn )
{
```

LISTING 5.2 Function **idft** used to calculate the inverse DFT of a complex array of any length. (*Continued*)

```
    // Used to store the coefficients in the complex vector
    static Vector<Complex> cf;
    // Stores n for future reference
    static int nstore = 0;

    // Check if empty vIn
    if( vIn.isEmpty() )
    {
        cf.empty();
        nstore = 0;
        return vIn;
    }

    int length = vIn.length();

    // Create return vector
    Vector<Complex> vOut = vIn;

    // Check if length changed from last time
    if( length != nstore )
    {
        // Clear previous coefficients and calculate new values
        cf.empty();
        nstore = length;

        // Create coefficients
        cf.setLength( length );

        double arg = 8.0 * atan( 1.0 ) / length;
        for( int i = 0; i < length; i++ )
        {
            cf[i].m_real = (float)( cos( arg * i )/(double)length );
            cf[i].m_imag = (float)( sin( arg * i )/(double)length );
        }
    }

    // Perform the DFT calculation
    for( int k = 0; k < length; k++ )
    {
        const Complex *dataIn = vIn.getData( length );
        vOut[k] = *dataIn++ * cf[0];
        for( int n = 1; n < length; n++ )
        {

            int p = ( ( n * k ) % length );
```

LISTING 5.2 (*Continued*)

```
            // Coefficient modulo length
            Complex cfm = cf[p];

            vOut[k] = vOut[k] + (*dataIn * cfm );

            // Go to next input sample
            dataIn++;
        }
    }
    return vOut;
}
```

LISTING 5.2 (*Continued*)

5.3 THE FAST FOURIER TRANSFORM ROUTINE

The radix two FFT computes the discrete Fourier transform in $N*\log_2(N)$ complex operations compared to N^2 complex operations for the direct DFT (where N is the transform length). This is a considerable speed-up and results in the almost exclusive use of the FFT for DFT calculation on computers. However, before the routine for the FFT is presented there are two drawbacks to its use as follows:

1. The radix 2 FFT only works on sequences with lengths that are a power of 2. In most cases, one can arrange to have a power of 2 length either by selecting the time window over which the data is observed or the sample rate at which data is taken, or by filling in a shorter array with zeroes to the required length. There are cases where none of these techniques work and a length other than a power of two must be used. In these cases, the DFT routine in Section 5.1 can be used.
2. The FFT has a certain amount of overhead (specifically bit-reversed ordering) that is unavoidable. This can make short-length FFT computation no faster than a straight DFT.

If neither of these drawbacks causes difficulty, then the FFT is one of the most efficient Fourier transform algorithms and is therefore the right tool for the frequency domain problem to be solved.

The basic element of an FFT is the *butterfly*. There are two ways to derive an algorithm for the FFT, called *decimation in time* (DIT) and *decimation in frequency* (DIF). As a result, there are two forms of the FFT butterfly that forms the basic part of the FFT, which is used many times in the algorithm. Flow graph diagrams of DIT and DIF forms for an 8-point FFT (with bit reversed input and normal order output) are shown in Figures 5.1(a) and 5.1(b), respectively. By reversing the flow graphs, a normal order input form of the FFT with bit-reversed output can be obtained as shown in Figure 5.2. The four

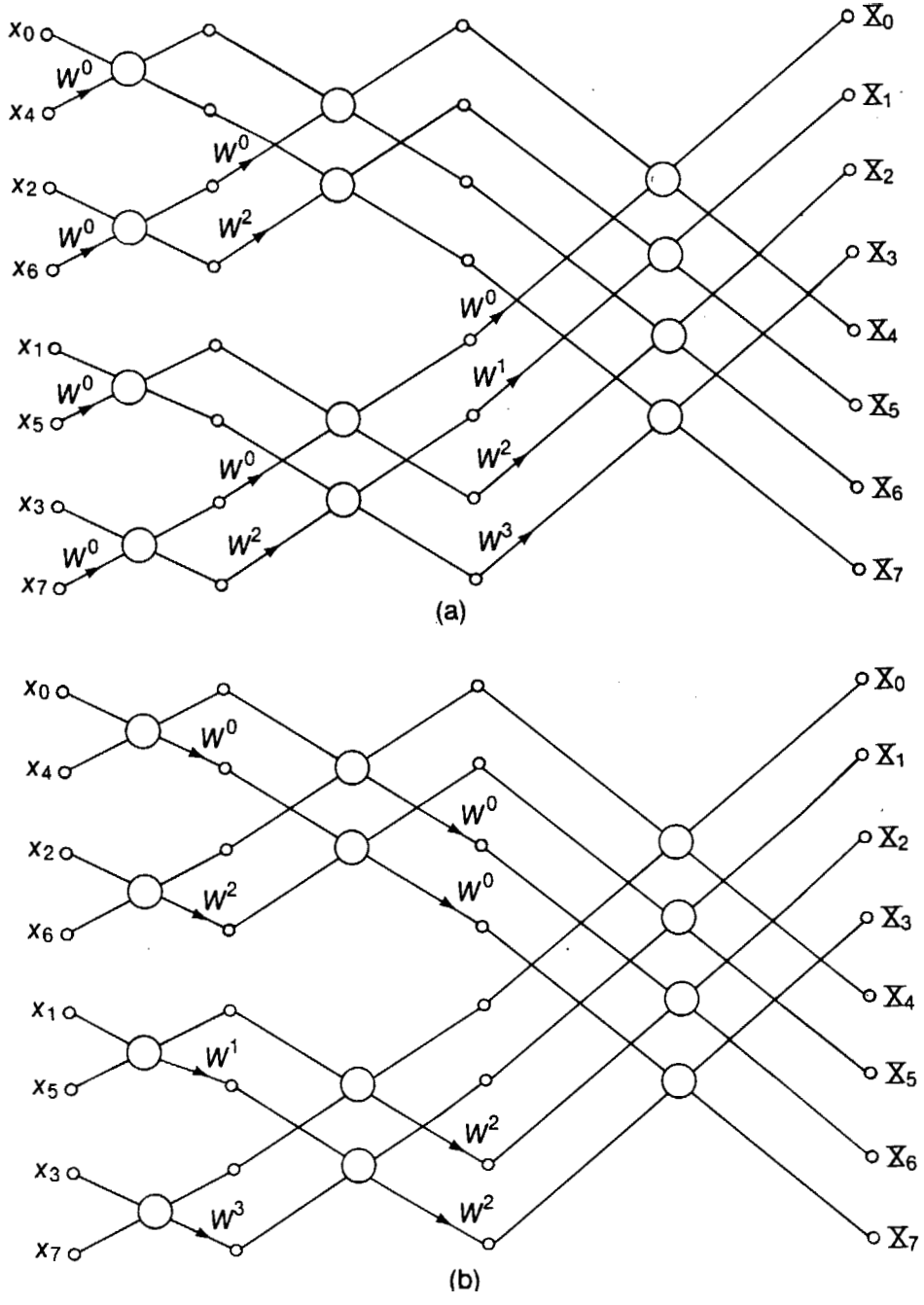


FIGURE 5.1 Flow graphs for 8-point FFTs with bit-reversed ordered inputs. Each flow graph has three passes of four butterflies each. (a) Decimation-in-time (DIT) flow graph. (b) Decimation-in-frequency (DIF) flow graph.

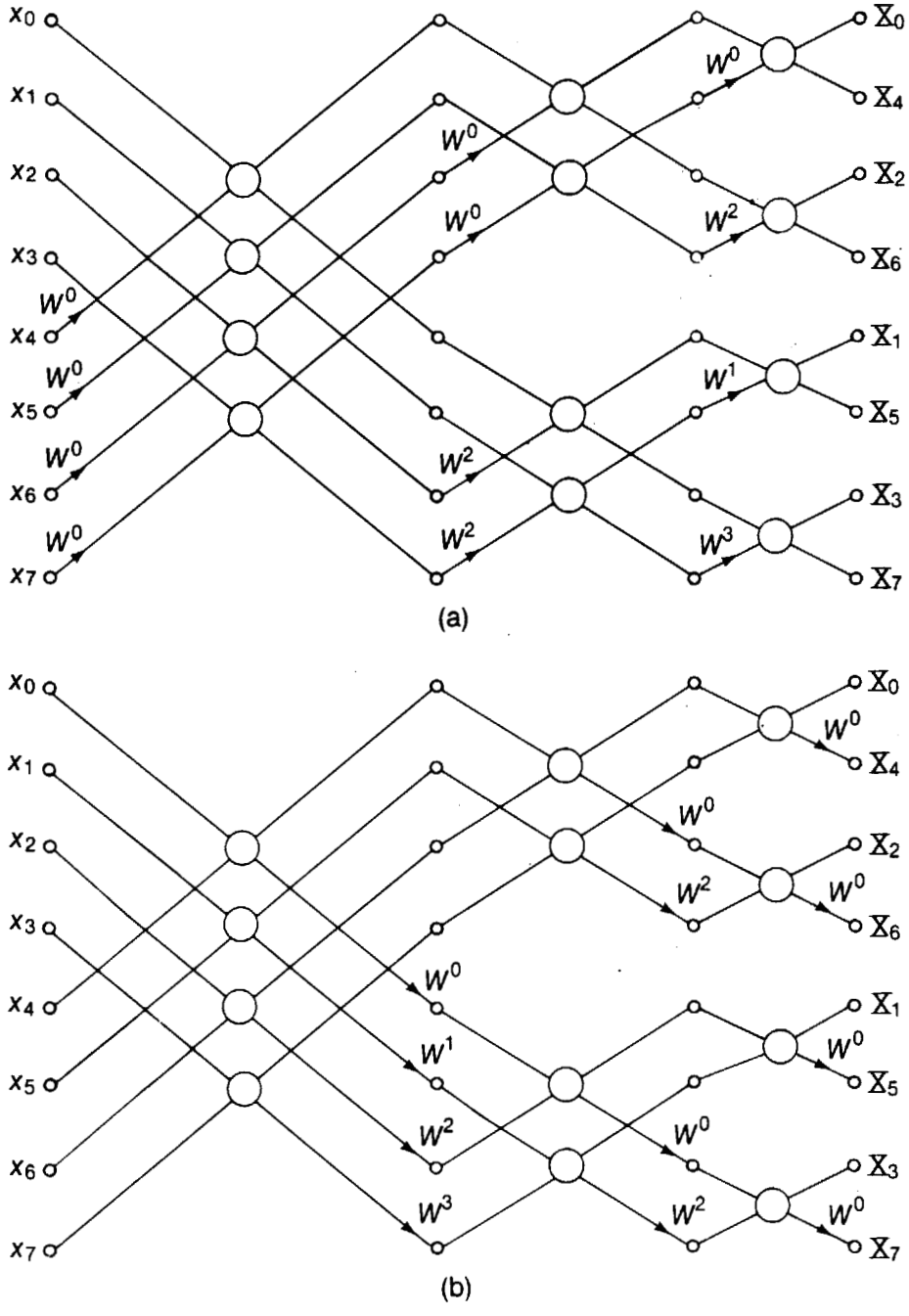


FIGURE 5.2 Flow graphs for 8-point FFts with normal-ordered inputs. Each flow graph has three passes of four butterflies each. (a) Decimation-in-time (DIT) flow graph. (b) Decimation-in-frequency (DIF) flow graph.

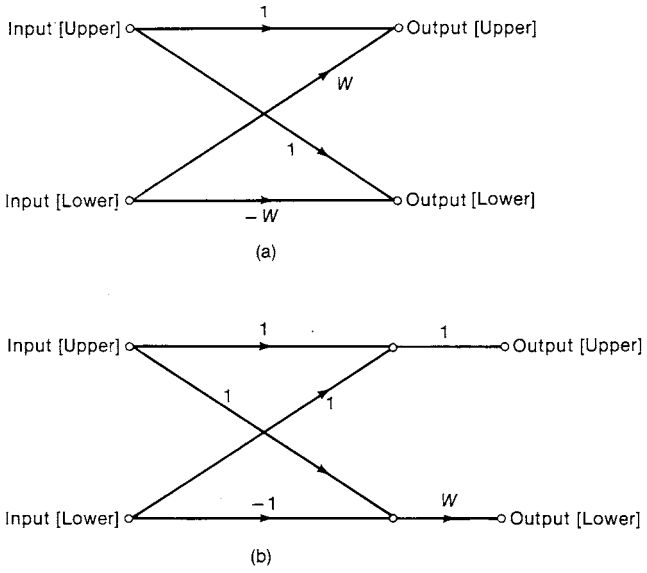


FIGURE 5.3 Flow graphs for single butterfly of the FFT. (a) Decimation in time (DIT) where the multiplication by the twiddle factor, *W,* is carried out before the additions and subtractions. (b) Decimation in frequency (DIF) where the additions and subtractions are done before the multiplication by the twiddle factor, *W.*

flow graphs are very similar, but the butterfly computation makes them a little different from each other. The two types of butterflies are extracted from the flow graphs and shown in detail in Figures 5.3(a) and 5.3(b). The equations for the decimation in time (DIT) butterfly are

```
output(upper) = input(upper) + W*input(lower)
output(lower) = input(upper) - W*input(lower).
```

The equations for the decimation in frequency (DIF) butterfly are

```
output(upper) =    input(upper) + input(lower)
output(lower) = W*[input(upper) - input(lower)].
```

The Cooley-Tukey FFT program, often cited in FFT literature, uses the DIF butterfly and performs the storage in-place. In-place algorithms are the most economical in terms of memory space, since the outputs of any given butterfly are placed in the storage from which the inputs were taken. In order to perform an in-place DIF butterfly, the following information is needed:

1. Index of the upper leg of the input
2. Difference between the upper and lower leg indices
3. Value of *W* (index into the stored *W* array)

To begin to develop the C++ routine for the FFT we will assign the following symbols to the variables:

x:	pointer to the beginning of the array of **COMPLEX** structures that will hold the input and output data
I:	index to the upper leg of the butterfly in the **x** array
le:	difference between the upper and lower leg indices
wptr:	a pointer to the current value of *W*
n:	the number of points in the FFT
m:	the log base 2 of **n**

Using these symbols and two temporary **COMPLEX** variables, **temp** and **tm,** the equations for the DIF butterfly can be rewritten as follows:

```
xi = x + i;
xip = xi + le;
temp = *xi + *xip;
tm = *xi - *xip;
*xip = tm * *wptr;
*xi = temp;
```

where the last statement transfers the contents of the temporary variable **temp** to the location pointed to by **xi** (the upper leg of the butterfly output).

Now that the butterfly calculation is developed, the next step is indicated by the DIF FFT flow graph, Figure 5.1(b). The butterflies are arranged in $\log_2(N)$ vertical stacks called *passes.* Within each pass the butterflies can be considered in *groups* distinguished by the pattern of *W* values used. In any pass, the first butterfly in each group will have the same value of *W,* the second in each group will have another value, and so on. We can take advantage of this structure by performing butterflies with the same value of *W* in the same loop. The following loop steps through the butterflies in a pass in order to accomplish this:

```
for (i = j ; i < n ; i = i + 2*le) {
      xi = x + i;
      xip = xi + le;
      temp = *xi + *xip;
      tm = *xi - *xip;
      *xip = tm * *wptr;
      *xi = temp;
}
```

In this loop the pointer to the upper leg begins at a value of **j** and steps by twice the distance between the upper and lower legs of the butterfly. It can be verified from the flow graph that butterflies spaced by this amount use the same *W* factor. The next element in the flow graph is the pass. Each pass is implemented by the following loop:

```
for (j = 0 ; j < le; j++) {
       for (i = j ; i < n ; i = i + 2*le) {
            xi = x + i;
            xip = xi + le;
            temp = *xi + *xip;
            tm = *xi - *xip;
            *xip = tm * *wptr;
            *xi = temp;
       }
       wptr = wptr + windex;
}
```

This loop steps the value of the index of the upper leg, **j,** through the upper leg positions of the first group in a pass. The inner loop takes care of all other groups using **j** as a starting point. The other function of the loop is to increment the pointer to the next *W* array value (**wptr**) by the **windex.** The value of **windex** changes with each pass.

The final step in implementing the FFT flow graph is to step through all **m** passes. The outer loop of the FFT routine does this as follows:

```
le = n;
windex = 1;
for (1 = 0 ; 1 < m ; 1++) {
       1e = 1e/2;
       wptr = w;
       for (j = 0 ; j < le ; j++) {
           for (i = j ; i < n ; i = 2*le) {
               xi = x + i;
               xip = xi + le;
               temp = *xi + *xip;
               tm = *xi - *xip;
               *xip = tm * *wptr;
               *xi = temp;
           }
           wptr = wptr + windex;
       }
       windex = 2*windex;
}
```

Note that **w** is the pointer to the beginning of the array of complex *W* twiddle factor values. Before each pass, the distance between the upper and lower legs of the butterflies, **le,**

is halved and the ***wptr** is initialized to the first twiddle factor. At the end of each pass, the value of **windex** is doubled.

The above program segment is the core of the FFT program. The complete FFT function, **fft,** is shown in Listing 5.3. Unlike the code segments discussed so far, the **fft** function code takes advantage of the fact that **w[0]** is equal to 1.0. This means that in the **for** loop involving **j,** in the first iteration (**j = 0**) when **wptr** has not been incremented by **windex,** no multiplications are needed, only additions and subtractions. The **fft** function takes advantage of this by creating a separate loop specifically for the **j = 0** case.

The program segment above assumed the **w** array was already generated. The **fft** code generates the **w** array recursively. This is a very efficient calculation method that avoids multiple calls to the C functions **cos** and **sin.** Another feature of the program is that **w** is generated as a **static** array and if the next invocation of **fft** uses the same FFT length (in other words, if **m** is equal to **mstore**), the **w** values are used without recalculation.

```
//////////////////////////////////////////////////////////////////
//
// fft( const Vector<Complex>& vIn, int lenFFT )
//   Performs in-place radix 2 decimation-in-time FFT.
//
//  Note:
//   vIn must be size power of 2.
//
//  Throws:
//   DSPException
//
//  Returns:
//   Vector<Complex> of transformed vector.
//
//////////////////////////////////////////////////////////////////
Vector<Complex> fft( const Vector<Complex>& vIn, int lenFFT )
{
    // Used to store the w complex array
    static Vector<Complex> w;
    // Stores m for future reference
    static int mstore = 0;
    // Length of FFT stored for future
    static int n = 1;

    // Check if empty vIn
    if( vIn.isEmpty() )
    {
```

LISTING 5.3 Function **fft** used to calculate the fast Fourier transform of a complex array with a power of 2 length. (*Continued*)

```
        w.empty();
        mstore = 0;
        n = 1;
        return vIn;
    }

    // Validate parameters
    int length = vIn.length();
    if( lenFFT == log2( length + 1 ) )
        throw DSPMathException( "FFT of non-power-of-two vector" );

    // Create return vector
    Vector<Complex> vOut = vIn;

    // Check if length changed from last time
    if( lenFFT != mstore )
    {
        // Clear previous w and calculate new values
        w.empty();
        mstore = lenFFT;

        // n = 2**m = fft length
        n = 1 << lenFFT;
        int le = n / 2;

        // Create w
        w.setLength( le - 1 );

        // Calculate the w values recursively

        // PI/le calculation
        double arg = 4.0 * atan( 1.0 ) / le;
        double wrecurReal = cos( arg );
        double wReal = wrecurReal;
        double wrecurImag = -sin( arg );
        double wImag = wrecurImag;

        for( int j = 0; j < le - 1; j++ )
        {
            w[j].m_real = (float)wrecurReal;
            w[j].m_imag = (float)wrecurImag;
            double wtempReal = wrecurReal * wReal - wrecurImag * wImag;
            wrecurImag = wrecurReal * wImag + wrecurImag * wReal;
            wrecurReal = wtempReal;
```

LISTING 5.3 (*Continued*)

```
        }
    }

    // Start FFT
    int le = n;
    int windex = 1;
    Complex u;
    Complex tm;
    Complex temp;

    for( int l = 0; l < lenFFT; l++ )
    {
        le = le / 2;

        // First iteration with no multiplies
        for( int i = 0; i < n; i = i + 2 * le )
        {
            temp = vOut[i] + vOut[i + le];
            vOut[i + le] = vOut[i] - vOut[i + le];
            vOut[i] = temp;
        }

        // Remaining iterations use stored w
        int wptr = windex - 1;
        for( int j = 1 ; j < le ; j++ )
        {
            u = w[wptr];
            for( i = j; i < n; i = i + 2 * le )
            {
                temp = vOut[i] + vOut[i + le];
                tm = vOut[i] - vOut[i + le];
                vOut[i + le] = tm * u;
                vOut[i] = temp;
            }
            wptr += windex;
        }
        windex *= 2;
    }

    // Rearrange data by bit reversing
    int j = 0;
    for( int i = 1; i < ( n - 1 ); i++ )
    {
        int k = n / 2;
        while( k <= j )
```

LISTING 5.3 (*Continued*)

```
        {
            j = j - k;
            k = k / 2;
        }
        j = j + k;
        if( i < j )
        {
            temp = vOut[j];
            vOut[j] = vOut[i];
            vOut[i] = temp;
        }
    }
    return vOut;
}
```

LISTING 5.3 (*Continued*)

The final section of **fft** is bit reversal. It can be seen from Figure 5.2(b) that the inputs to the DIF flow graph are normally ordered from top to bottom of the page. At the output, however, the order of the frequency components is scrambled. In a radix 2 FFT this scrambling is always in an order known as *bit reversed.* This term means that if the output position indices were represented in binary format, listed in order, and only just enough bits were used to represent the highest index (**m** bits), then the output order of the FFT could be determined by reading the position indices in backward bit order: 00101011 (position 75 of a 256-point FFT) would become 11010100, representing output number 212.

The **fft** function performs bit reversal in a method adapted from the Cooley-Tukey algorithm. The following code segment also performs bit reversal but uses a method taking advantage of the Boolean capabilities of C/C++:

```
for (i = 0 ; i < n/2 ; i++){
    j = 0;
    for (k=0; k<m; ++k)
        j = (j << 1) | (1 & (i >> k));
    xi = x + i;
    xj = x + j;
    temp = *xj;
    *xj = *xi;
    *xi = temp;
}
```

Although this second method is slightly slower than the Cooley-Tukey code, it makes clear why the ordering is referred to as bit reversed: The bits from the index **i** are successively added to the index **j** in reverse order to create a final bit-reversed index.

5.4 THE INVERSE FFT ROUTINE

As with the DFT and inverse DFT, the inverse FFT is essentially the same program as the forward FFT, except the complex conjugate of the coefficients are used and the $1/N$ scaling is performed at the end of the routine. Listing 5.4 shows the source code for the **ifft** function, which performs the inverse radix 2 fast Fourier transform on complex time domain data. The algorithm is in-place, so that the input data is overwritten by the output data.

```
//////////////////////////////////////////////////////////////////////
//
// ifft( const Vector<Complex>& vIn, int lenFFT )
//   Performs in-place radix 2 decimation-in-time inverse FFT.
//
//  Note:
//   vIn must be size power of 2.
//
//  Throws:
//   DSPException
//
//  Returns:
//   Vector<Complex> of transformed vector.
//
//////////////////////////////////////////////////////////////////////
Vector<Complex> ifft( const Vector<Complex>& vIn, int lenFFT )
{
    // Used to store the w complex array
    static Vector<Complex> w;
    // Stores m for future reference
    static int mstore = 0;
    // Length of FFT stored for future
    static int n = 1;

    // Check if empty vIn
    if( vIn.isEmpty() )
    {
        w.empty();
        mstore = 0;
        n = 1;
        return vIn;
    }
```

LISTING 5.4 Function **ifft** used to calculate the inverse fast Fourier transform of a complex array with a power of 2 length. (*Continued*)

```
    // Validate parameters
    int length = vIn.length();
    if( lenFFT == log2( length + 1 ) )
        throw DSPMathException( "FFT of non-power-of-two vector" );

    // Create return vector
    Vector<Complex> vOut = vIn;

    // Check if length changed from last time
    if( lenFFT != mstore )
    {
        // Clear previous w and calculate new values
        w.empty();
        mstore = lenFFT;

        // n = 2**m = fft length
        n = 1 << lenFFT;
        int le = n / 2;

        // Create w
        w.setLength( le - 1 );

        // Calculate the w values recursively

        // PI/le calculation
        double arg = 4.0 * atan( 1.0 ) / le;
        double wrecurReal = cos( arg );
        double wReal = wrecurReal;
        // Opposite sign from FFT
        double wrecurImag = sin( arg );
        double wImag = wrecurImag;

        for( int j = 0; j < le - 1; j++ )
        {
            w[j].m_real = (float)wrecurReal;
            w[j].m_imag = (float)wrecurImag;
            double wtempReal = wrecurReal * wReal - wrecurImag * wImag;
            wrecurImag = wrecurReal * wImag + wrecurImag * wReal;
            wrecurReal = wtempReal;
        }
    }

    // Start IFFT
    int le = n;
    int windex = 1;
```

LISTING 5.4 (*Continued*)

```
    Complex u;
    Complex tm;
    Complex temp;

    for( int l = 0; l < lenFFT; l++ )
    {
        le = le / 2;
        // First iteration with no multiplies
        for( int i = 0; i < n; i = i + 2 * le )
        {
            temp = vOut[i] + vOut[i + le];
            vOut[i + le] = vOut[i] - vOut[i + le];
            vOut[i] = temp;
        }

        // Remaining iterations use stored w
        int wptr = windex - 1;
        for( int j = 1 ; j < le ; j++ )
        {
            u = w[wptr];
            for( i = j; i < n; i = i + 2 * le )
            {
                temp = vOut[i] + vOut[i + le];
                tm = vOut[i] - vOut[i + le];
                vOut[i + le] = tm * u;
                vOut[i] = temp;
            }
            wptr += windex;
        }
        windex *= 2;
    }

    // Rearrange data by bit reversing
    int j = 0;
    for( int i = 1; i < ( n - 1 ); i++ )
    {
        int k = n / 2;
        while( k <= j )
        {
            j = j - k;
            k = k / 2;
        }
        j = j + k;
        if( i < j )
        {
```

LISTING 5.4 (*Continued*)

```
            temp = vOut[j];
            vOut[j] = vOut[i];
            vOut[i] = temp;
        }
    }

    // Scale all results by 1/n
    float scale = (float)( 1.0f / n );
    for( i = 0; i < n; i++ )
    {
        vOut[i] = scale * vOut[i];
    }
    return vOut;
}
```

LISTING 5.4 (*Continued*)

5.5 WINDOWING ROUTINES

All windowing routines have as their purpose the reduction of the side lobes of a spectral output of the FFT or DFT routines. They accomplish this by forcing the beginning and end of any sequence to approach each other in value. Since they must work with any sequence, they force the beginning and ending samples near zero. To make up for this reduction in power, windowing routines give extra weight to the values near the middle of the sequence. The difference between windows is the way in which they transition from the low weights near the edges to the higher weights near the middle of the sequence. The basic routine for windowing is a simple real-time complex vector multiply as follows:

```
float *h, *hptr;
float *x, *xi;
xi = x;
hptr = h;
for (i = 0; i < n; i++){
   *xi++ *= *hptr++;
}
```

The windowing weights are stored in the vector of values beginning at the pointer **h.** These are real numbers (**float**s), which are used to scale both the real and imaginary parts of the **COMPLEX** sequence at **x.** Both the sequence and the window must be **n** samples long. Examples of routines that create windows are as follows (**PI** is defined previously to be 3.14159 . . .):

Hamming window:

```
hptr = h;
arg = 2.0*PI/(n-1);
for (i = 0 ; i < n ; i++)
    *hptr++ = 0.54 - 0.46*cos(arg*i);
```

Hanning window:

```
hptr = h;
arg = 2.0*PI/(n-1);
for (i = 0 ; i < n ; i++)
    *hptr++ = 0.5 - 0.5*cos(arg*i);
```

Bartlett window (triangle):

```
hptr = h;
a = 2.0/(n-1);
for (i = 0 ; i <= (n-1)/2 ; i++)
    *hptr++ = i*a;
for ( ; i < n ; i++)
    *hptr++ = 2.0 - i*a;
```

Blackman window (3 term):

```
hptr = h;
arg = 2.0*PI/(n-1);
for (i = 0 ; i < n ; i++)
    *hptr++ = 0.42 - 0.5*cos(arg*i) + 0.08*cos(2*arg*i);
```

Blackman-Harris window (4 term):

```
hptr = h;
arg = 2.0*PI/(n-1);
for (i = 0 ; i < n ; i++)
    *hptr++ = 0.35875 - 0.48829*cos(arg*i) +
       0.14128*cos(2*arg*i) - 0.01168*cos(3*arg*i);
```

If multiple sequences are to be windowed and transformed, a stored array (as shown above) is the most efficient method of windowing the input sequences. If a single sequence is to be windowed, the generation and windowing can be combined allowing storage space to be saved as follows for the Hamming window:

```
arg = 2.0*PI/(n-1);
xi = x;
for (i = 0 ; i < n ; i++) {
    wvalue = 0.54 - 0.46*cos(arg*i);
    xi->real *= wvalue;
```

```
        xi->imag *= wvalue;
        xi++;
    }
```

The set of five functions contained in the DFT.CPP source code file uses this last technique to perform windowing on **COMPLEX** input arrays. The complex input array is multiplied by the window function such that the result overwrites the input array. The function **ham,** which implements the Hamming window, is as follows:

```
////////////////////////////////////////////////////////////////
//
// ham( const Vector<Complex>& vIn )
//   Performs Hamming window on complex vector.
//
//  Returns:
//   Vector<Complex> of windowed vector.
//
////////////////////////////////////////////////////////////////
Vector<Complex> ham( const Vector<Complex>& vIn )
{
    Vector<Complex> vOut = vIn;
    if( vOut.isEmpty() )
        return vOut;

    double factor = 8.0 * atan( 1.0 ) / ( vOut.length() - 1 );
    for( int i = 0; i < vOut.length(); i++ )
    {
        double ham = 0.54 - 0.46 * cos( factor * i );
        vOut[i] *= ham;
    }
    return vOut;
}
```

In this function, the variable factor is the same as **2*PI/(n − 1)** calculated using the **atan** function and can be replaced with a constant for greater efficiency. The functions **han, triang, black,** and **harris** implement the Hanning, triangular, 3-term Blackman, and 4-term Blackman-Harris windows, respectively, in a similar fashion.

5.6 MAGNITUDE, PHASE, AND LOGARITHMIC DISPLAYS

The results of a DFT or an FFT on a sequence are most often displayed as the magnitude of the complex numbers as a function of frequency sample number. Information is lost in this type of display, since the phase of the frequency sample is discarded. If the phase information is important, it can be displayed in a separate graph. Very often it is most informative to present the logarithm of the magnitudes, since this shows the behavior of the signal at low amplitudes as well as higher amplitudes. In the case of a filter transfer func-

tion, the logarithmic display shows the stopband characteristic in more detail than a linear magnitude display.

To generate the power spectrum or periodogram of an FFT result the following code segment can be used:

```
xi = x;
for (i=0; i<n; i++){
   xi->real *= xi->real;
   xi->real += xi->imag * xi->imag;
   xi++;
}
```

The real portion of the array **x** will hold the power spectrum after this routine. If the real FFT result must be saved, then another array must be declared and the power spectrum placed in it. To create the log magnitude display, the following code segment can be used:

```
xi = x;
for (i=0; i<n; i++){
   xi->real *= xi->real;
   xi->real += xi->imag * xi->imag;
   xi->real = 10*log10((double) xi->real);
   xi++;
}
```

The factor of 10 is used in the statement involving **log10** in order to make the units of the result decibels (dB).

There are times when the phase of the signal is also important. The following code combines the calculation of power magnitude, log magnitude, and phase. The magnitude is placed in the real part of the input array, the phase in the imaginary part, and the log magnitude in a separate array: **lmag** (which must be allocated previously).

```
COMPLEX *x, *xi, phase;
float *lmag, *lptr;
int i;
xi = x;
lptr = lmag;
for (i=0; i<n; i++){
   phase = atan2(xi->imag,xi->real);
   xi->real * = xi->real;
   xi->real + = xi->imag * xi->imag;
   xi->imag = phase;
   lptr = 10*log10(MAX(1.e-14,xi->real));
   xi++;
   lptr++;
}
```

In the above code, the **max** macro (described in Chapter 2 and defined in file GET.H) is used to clip the logarithmic output at -140 dB. Without this clipping, zero magnitude values could cause a floating-point underflow.

5.7 OPTIMIZING THE FFT FOR REAL INPUT SEQUENCES

The FFT is an excellent tool for spectrum analysis. It is so universal that it will even directly determine the spectral character of the complex input sequences generated by RADAR, ultrasound, and communications systems where coherent signal demodulators are used. It seems a waste to use such a powerful tool when the input sequence is often strictly real as in SONAR and speech analysis. As a result, two ways to take advantage of the full power of the FFT while processing strictly real input sequences have been developed. The first method, called *trigonometric recombination,* is described in this section. This method accepts any real input sequence and processes it using an FFT of half the size normally required for a complex sequence. This results in nearly a factor of two speed-up in the computation.

The second method, called the *double-sequence FFT,* will not be discussed in detail. This algorithm accepts two real sequences of length *N* and processes them using a single *N*-point complex FFT. This is primarily useful in real-time systems where continuous data is being transformed. The choice of which algorithm to use is based upon the processing problem to be solved. Trigonometric recombination does not require two sequences, but the recombination processes take more computations and time than the double-sequence FFT method. If two or more identical-length FFTs are to be performed as part of the problem, the double-sequence FFT is a slightly better choice.

A derivation of the trigonometric recombination method will not be presented here, because many of the references on the FFT at the end of this chapter contain good expositions. The technique of recombination is applied after a *N*/2 length complex FFT has been performed on a strictly real input sequence of length *N.* Before the FFT is performed, the input sequence is split into a "real" part consisting of the even elements of the original sequence, and an "imaginary" part consisting of the odd elements of the original.

The complex FFT of length *N*/2 is now performed on the sequence **x.** It is clear that the result will not be the complex spectrum of the real sequence **a.** There is a set of recombination computations that must be performed after the FFT to obtain the desired result. The C statements for recombination using the complex spectrum (now stored in the **x** array) can be written as follows for index **k** running from 1 to *N*/2 - 1:

```
Realsum[k] = ( x[k].real + x[N/2 - k].real ) / 2;
Imagsum[k] = ( x[k].imag + x[N/2 - k].imag ) / 2;
Realdif[k] = ( x[k].real - x[N/2 - k].real ) / 2;
Imagdif[k] = ( x[k].imag - x[N/2 - k].imag ) / 2;

b[k].real = Realsum[k] + cos(2*PI*k/N) * Imagsum[k]
          - sin(2*PI*k/N) * Realdif[k]
```

```
b[k].imag = Imagdif[k] - sin(2*PI*k/N) * Imagsum[k]
          - cos(2*PI*k/N) * Realdif[k];
```

For the $k = 0$ case (the DC term) the result is calculated without any multiplies as follows:

```
b[0].real = x[0].real + x[0].imag;
b[0].imag = 0.0;
```

Because of the sums and differences are computed before a complex multiply by fixed coefficients, these computations look very similar to a decimation in time butterfly. Most special-purpose hardware for real input FFTs take advantage of this similarity and perform trigonometric recombination as essentially one more pass in the FFT computation.

So far we have not shown the formula for values of k greater than $N/2 - 1$. If a complex FFT is provided with a real input signal, the frequency components above $N/2$ represent the negative half of the frequency spectrum. Thus, the k values greater than $N/2$ represent the negative frequency spectrum, which is the complex conjugate of the positive frequency spectrum. For all values of k greater than $N/2$ (the *midpoint* or *Nyquist point* of the spectrum) the spectral components can be determined from the other values as follows:

```
x[k].real = x[N - k].real;
x[k].imag = -x[N - k].imag;
```

Due to this conjugate symmetry, the magnitude values are a mirror image around the midpoint of the spectrum of a real sequence. Our real-input FFT routine using trigonometric recombination will return only the first half of the spectrum. This saves storage and allows an in-place implementation for most of the calculations while still providing a complete spectrum. The complete real-input FFT using the trigonometric recombination algorithm is shown in Listing 5.5. A complex FFT of length N would normally require $(N/2)*\log_2(N)$ complex multiplies and $(N/2)*\log_2(N)$ complex adds. The trigonometric recombinations done after the FFT requires $N/2$ complex multiplies for an N length real sequence. Therefore, the real FFT requires $N/2 + (N/4)*\log_2(N/2)$ complex multiplies because the complex FFT length is half the real input length. Thus, the time required by the real FFT is approximately half of the time required by a similar-size complex FFT.

5.8 FOURIER TRANSFORM EXAMPLES

All the functions needed for the examples in this chapter have now been presented. The example programs presented in this section are all written to accept data from files on disk in the **DSPFile** format discussed in Chapter 3 and to write their results to files in this same format. The WINPLOT program provided on the disk (see Chapter 3, Section 3.3) can then be used to display the data. By studying the example program code, the method of incorporating the functions into larger application programs can be seen.

```
//////////////////////////////////////////////////////////////////////
//
// rfft( const Vector<float> vIn )
//   Performs trigonometric recombination real input FFT.
//   The FFT will be the length of vIn and the output
//   vector will be the lower half of the elements
//   of the spectrum ( N / 2 + 1 ).
//
//
//  Note:
//   vIn must be size power of 2.
//
//  Throws:
//   DSPException
//
//  Returns:
//   Vector<Complex> of transformed vector.
//
//////////////////////////////////////////////////////////////////////
Vector<Complex> rfft( const Vector<float> vIn )
{
    // Stored coefficients
    static Vector<Complex> vCF;

    // Get length of FFT
    int lenFFT = log2( vIn.length() );
    if( vIn.length() != 1 << lenFFT )
        throw DSPMathException( "RFFT of non-power-of-two vector" );

    // Convert real input to Complex pairs
    Vector<Complex> vX( vIn.length() / 2 );
    for( int i = 0; i < vX.length(); i++ )
    {

        // Real value is even input value
        vX[i].m_real = vIn[2 * i];
        // Imag value is odd input value
        vX[i].m_imag = vIn[( 2 * i ) + 1];
    }

    // Call FFT with half the size of the real FFT
    Vector<Complex> vCX = fft( vX, lenFFT - 1 );

    // Create the coefficients for recombination
```

LISTING 5.5 Function **rfft** used to calculate the fast Fourier transform of a real array with a power of 2 length. (*Continued*)

```
    int num = vCX.length();
    if( vCF.length() != num - 1 )
    {
        vCF.empty();
        vCF.setLength( num - 1 );
        double factor = 4.0 * atan( 1.0 ) / num;
        for( int k = 1; k < num; k++ )
        {
            double arg = factor * k;
            vCF[k-1].m_real = (float)cos( arg );
            vCF[k-1].m_imag = (float)sin( arg );
        }
    }

    // Output vector
    Vector<Complex> vOut( num );

    // DC component, no multiplies
    vOut[0].m_real = vCX[0].m_real + vCX[0].m_imag;
    vOut[0].m_imag = (float)0.0;

    // Other frequencies by trig recombination

    // Indices have different offsets
    int ck = 0;
    int xk = 1;
    int xnk = num - 1;
    for( int k = 1; k < num; k++, ck++, xk++, xnk— )
    {
        float realSum = ( vCX[xk].m_real + vCX[xnk].m_real ) / 2;
        float imagSum = ( vCX[xk].m_imag + vCX[xnk].m_imag ) / 2;
        float realDiff = ( vCX[xk].m_real - vCX[xnk].m_real ) / 2;
        float imagDiff = ( vCX[xk].m_imag - vCX[xnk].m_imag ) / 2;

        double tmp = 0.0;
        tmp = realSum + vCF[ck].m_real * imagSum;
        tmp -= vCF[ck].m_imag * realDiff;
        vOut[k].m_real = (float)tmp;

        tmp = imagDiff - vCF[ck].m_imag * imagSum;
        tmp -= vCF[ck].m_real * realDiff;
        vOut[k].m_imag = (float)tmp;
    }
    return vOut;
}
```

LISTING 5.5 (*Continued*)

5.8.1 FFT Test Routine

This program illustrates the use of the **fft** function, the windowing functions, and the log magnitude code. The source code for the FFTTEST program is shown in Listing 5.6. The program accepts a formatted DSP input file, windows the data using a window function chosen by the user, and then performs an FFT on the windowed time domain data. The log of the magnitude of the power spectrum is then calculated and written to another disk file.

```
////////////////////////////////////////////////////////////////////////
//
// ffttest.cpp - Test FFT and window functions
//
////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "dft.h"
#include "disk.h"
#include "get.h"

// Maximum FFT size (2^n)
const int MAXFFTSIZE = 12;

////////////////////////////////////////////////////////////////////////
//
// int main()
//   Tests FFT and window functions. Accepts time
//   domain signal input file. Gives spectral magnitude
//   file and time domain windows display as output.
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Open the input file
        do getInput( "Enter input signal file", strName );
```

LISTING 5.6 Program FFTTEST used to demonstrate the fast Fourier transform and windowing functions. (*Continued*)

```
while( strName.isEmpty() || dspfile.isFound( strName ) == false );
dspfile.openRead( strName );

// Read vector from file
Vector<float> vSignal;
dspfile.read( vSignal );
dspfile.close();

// Get length of FFT
int lenIn = vSignal.length();
int m = MAXFFTSIZE;
if( log2( lenIn ) < MAXFFTSIZE )
{
    getInput(
        "Enter power of 2 length of FFT",
        m,
        log2( lenIn ),
        MAXFFTSIZE );
}
else
{

    cout
        << "Warning: Truncating signal to "
        << ( 1 << MAXFFTSIZE ) << " samples for FFT\n";
}

// Get the type of window to use on the time domain data
cout << "The available window functions are:\n";
cout << " 1 -- Rectangular\n";
cout << " 2 -- Hamming\n";
cout << " 3 -- Hanning\n";
cout << " 4 -- Triangle\n";
cout << " 5 -- Blackman\n";
cout << " 6 -- 4 term Blackman-Harris\n";

int wnum = 0;
getInput( "Enter the number of the window function", wnum, 1, 6);

// Perform the window function requested by the user
Vector<Complex> vWin;
conv( vWin, vSignal);
switch( wnum )
{
```

LISTING 5.6 (*Continued*)

```
case 1:
    // Rectangle (None)
    break;
case 2:
    vWin = ham( vWin );
    break;
case 3:
    vWin = han( vWin );
    break;
case 4:
    vWin = triang( vWin );
    break;
case 5:
    vWin = black( vWin );
    break;
case 6:
    vWin = harris( vWin );
    break;
}

// Allocate vector large enough to hold data (padded to 2^n)
int lenFFT = 1 << m;
Vector<Complex> vSamp( lenFFT );
vSamp = (Complex)0.0f;

// Copy windowed data and write it to file
for( int i = 0; i < lenIn; i++ )
{
    vSamp[i] = vWin[i];
    vSignal[i] = vWin[i].m_real;
}

// Get windowed signal file
do getInput( "Enter windowed signal file to create", strName );
while( strName.isEmpty() );
dspfile.openWrite( strName );
dspfile.write( vSignal );

// Make descriptive trailer for windowed signal file
String strTrailer;
getInput( "Enter trailer string", strTrailer );

if( strTrailer.isEmpty() == false )
    dspfile.setTrailer( strTrailer );
dspfile.close();
```

LISTING 5.6 (*Continued*)

```
// Find the spectrum of the data
vSamp = fft( vSamp, m );

// Do log magnitude
double logMag = 4.0 / ( lenIn * lenIn );
for( i = 0; i < lenFFT ; i++ )
{
    double tempFilt = vSamp[i].m_real * vSamp[i].m_real;
    tempFilt += vSamp[i].m_imag * vSamp[i].m_imag;
    tempFilt *= logMag;
    vSamp[i].m_real = (float)( 10*log10( max( tempFilt, 1.e-14 )));
}

// Copy a section of the spectrum to the magnitude file
// to allow a close up plot of a particular segment of the
// spectral magnitude of the signal.
int center = 0;
if( lenFFT > 1 )
    getInput(
        "Enter center of magnitude file",
        center,
        0,
        lenFFT - 1 );
int view = 10;
if( lenFFT > 10 )
    getInput(
        "Enter length of magnitude file",
        view,
        10,
        lenFFT );
int begin = center - ( view / 2 );

Vector<float> logMagView( view );

// Format logMagView
for( int k = 0; k < view; k++ )
{
    i = k + begin;
    if( i < 0 )
        i = 0;
    if( i >= lenFFT )
        i = lenFFT - 1;
    logMagView[k] = vSamp[i].m_real;
}
```

LISTING 5.6 (*Continued*)

```
        // Write magnitude to file
        do getInput( "Enter magnitude file to create", strName );
        while( strName.isEmpty() );
        dspfile.openWrite( strName );
        dspfile.write( logMagView );

        // Make descriptive trailer for magnitude file
        getInput( "Enter trailer string", strTrailer );

        if( strTrailer.isEmpty() == false )
            dspfile.setTrailer( strTrailer );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 5.6 (*Continued*)

A function called **log2** is used in FFTTEST to create the number of passes in the FFT that is the logarithm in base 2 of the FFT length (passed to the **fft** function as the second argument). This function is used in several other routines in this chapter and is shown in Listing 5.7. The function accepts an unsigned integer **x** as its single parameter and returns an **int.** The unsigned integer parameter allows inputs up to 65535 on machines where the integer is 16 bits. In the function, an integer **i** is incremented as successive shifts are performed on a masking value (**mask**) until the mask causes the entire value of **x** to be ANDed with zero. When **x** has been completely masked, the value of **i** is the log base 2 of **x.**

To show the envelopes of the windows provided by the window functions in the DFT.CPP file, a simple input is created at a relative frequency of 0.05. This allows an uncorrupted picture of the windowing function in the time domain. The program MKWAVE (see Chapter 3, Section 3.3) is used to create the example input file (called FREQ_05.DAT) as follows:

```
Enter number of samples to generate [2..10000] : 1024
Enter number of frequencies in the sum [1..20] : 1
Enter frequency #0 [0.0..0.5] : .05
Enter file name : FREQ_05.DAT
```

```
//////////////////////////////////////////////////////////////////
//
// log2( int x )
//   Calculate base 2 logarithm of x.
//
//  Returns:
//   Base 2 log of x rounded to the higher integer.
//
//////////////////////////////////////////////////////////////////
inline int log2( int x )
{
    // Validate parameters
    if( x <= 0 )
    {
        // Cannot have the log of 0
        throw DSPMathException( "Log2 of zero." );
    }

    // Get the max index —- x - 1
    x—;
    for( int i = 0; x; i++ )
        x >>= 1;
    return i;
}
```

LISTING 5.7 Inline function **log2** used to determine the base 2 logarithm of and integer (contained in file DSP.H).

```
Signal of length 1024 equal to the sum of 1
cosine waves at the following frequencies:
f/fs = 0.05
```

The resulting file can be used as input for the program FFTTEST. In order to show the side-lobe performance of the windows, a 4096-point FFT is run on the 1024-point sequence. The computer dialogue using FFTTEST employing the first window selection (a *rectangular* or *Dirichlet window*) is as follows:

```
Enter the input signal file name : FREQ_05.DAT
Enter the power of 2 length of the FFT [10..12] : 12
The available window functions are:
          1 — Rectangular
          2 — Hamming
          3 — Hanning
          4 — Triangle
          5 — Blackman
```

```
          6 – 4 term Blackman-Harris
Enter the number of the window function desired [1..6] : 1
Enter windowed signal file name : RECT.WIN
Enter trailer spring : Rectangular windowed input.
Enter center of magnitude file [0..1023] : 200
Enter length of magnitude file [10..1024] : 400
Enter spectral magnitude file name : RECT.MAG
Enter trailer string : Spectral magnitude of freq_05.dat.
```

The results of this dialogue are now plotted. The plots of the RECT.WIN and RECT.MAG files are shown in Figure 5.4. Figure 5.4(a) is the input file created using MKWAVE, in this case because the windowed file is identical to the input, Figure 5.4(b) is the spectral magnitude file. Note the side lobes due to rectangular windowing.

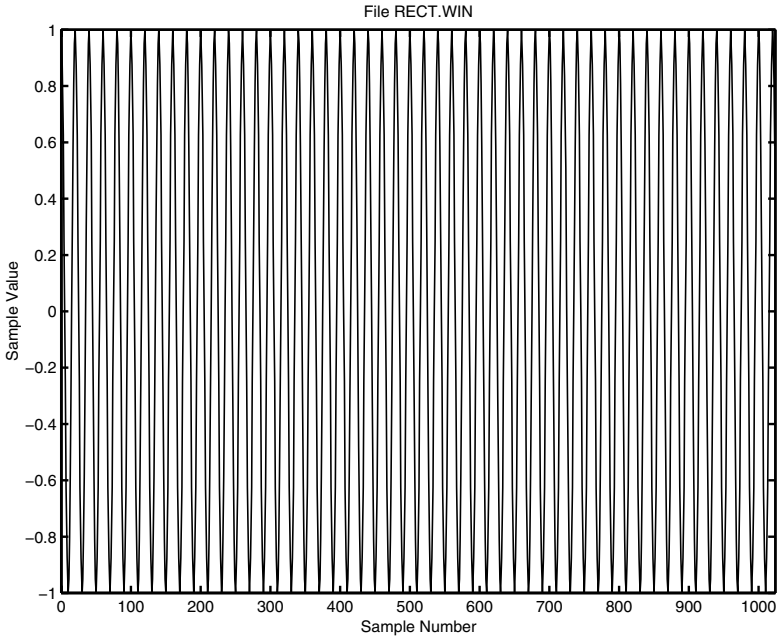
Next FFTTEST is used to create five more example outputs, each with a different window. Figures 5.5 through 5.9 show the windowed time-domain data and the power spectrum for the Hamming, Hanning, triangle (Bartlett), Blackman, and Blackman-Harris windows, respectively. The paper by Harris referenced at the end of this chapter is a very authoritative source for relative window performance. An excerpt of a table from this paper is shown in Table 5.1. This table shows how the window functions compare in performance in terms of side-lobe level (expressed in dB relative to the maximum response at DC) and bandwidth (expressed in frequency bins of a DFT of the same length as the window).

The four-term Blackman-Harris is the best at reducing side-lobe level (spectral leakage), but its equivalent noise bandwidth is among the largest. This shows up in the examples as the width of the main lobe of the spectrum. The rectangular window has the smallest equivalent bandwidth but also the worst spectral leakage (highest side-lobe level). The Hamming window is an excellent compromise between the two extremes and is probably the most popular and widely used general-purpose window function.

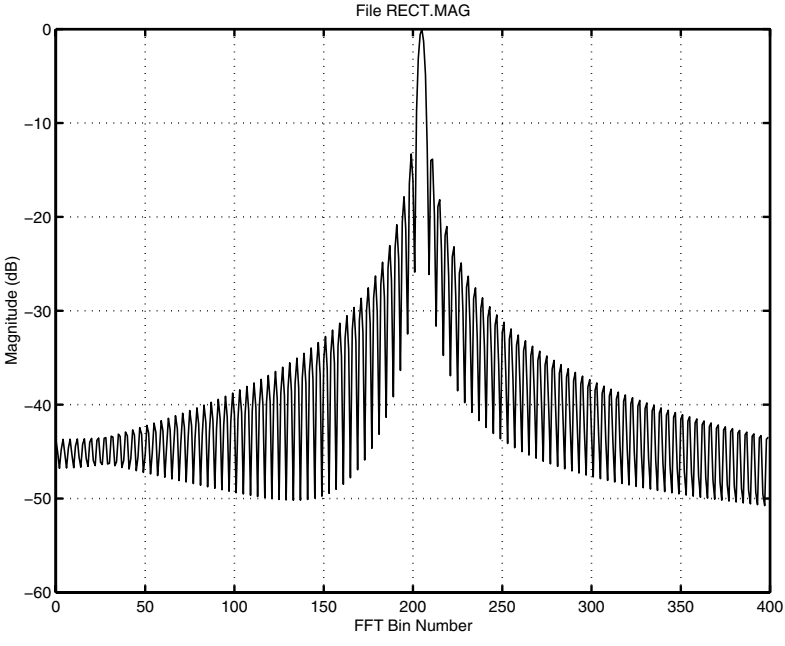
5.8.2 DFT Test Routine

This program illustrates the use of the **dft** and **idft** functions in a program that accepts a time-domain input from disk, and creates both the log magnitude of the power spectrum and a reconstructed version of the original signal file. The source code is shown in

FIGURES 5.4 TO 5.9 Graphs of the outputs of the program FFTTEST using the data file FREQ_0.5.DAT as input. This input file consists of 1024 samples of single cosine waveform of relative frequency 0.05. For each of the six windows available in the program FFTTEST, the windowed input and the resulting spectral magnitude are graphed. The input frequency was chosen so that the shape of the windows in the time domain can be compared. The relative performance of the windows can be compared by height of the first side lobes (care must be taken here, since the magnitude scales are different) and the magnitude and width of the main lobe. The size of the side lobes is a measure of spectral leakage. The main lobe characteristic width is a measure of processing loss due to the window.

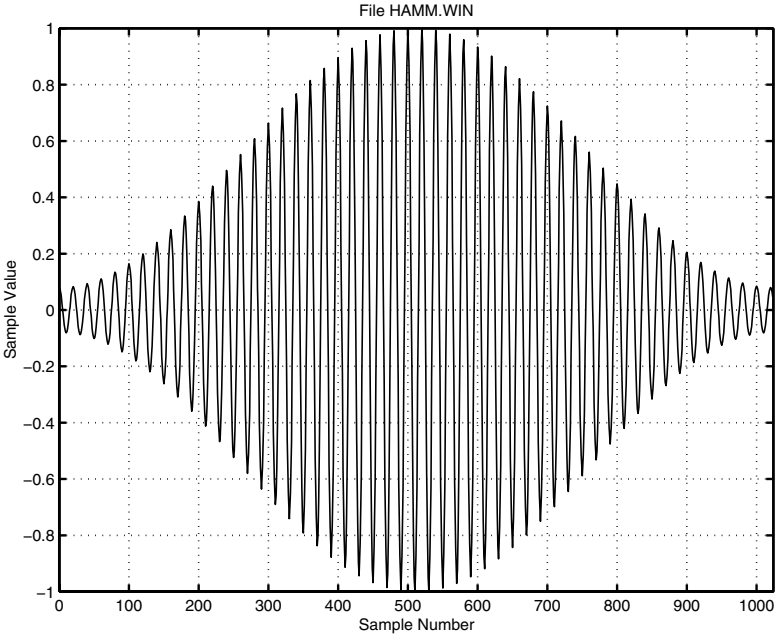


(a)

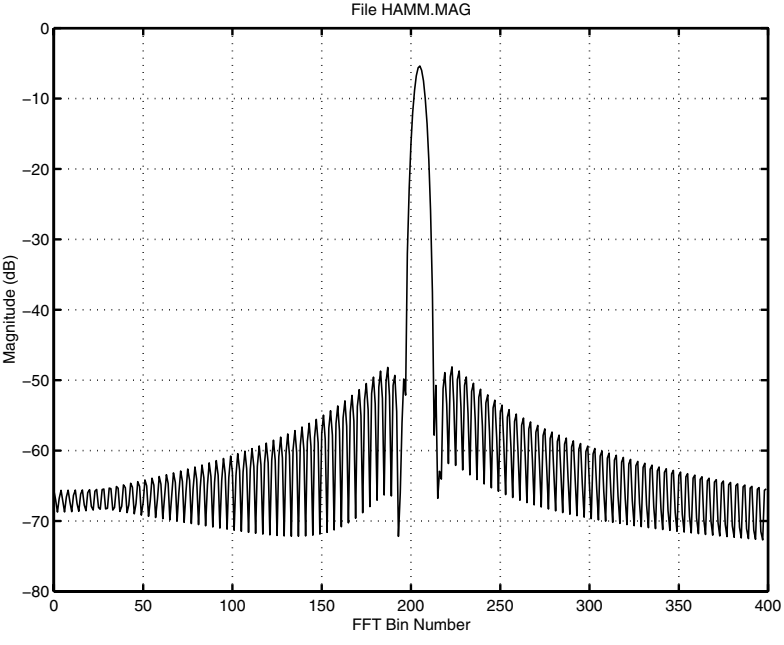


(b)

FIGURE 5.4 (a) Input waveform (also, rectangular windowed input). (b) Spectral magnitude using a rectangular window.

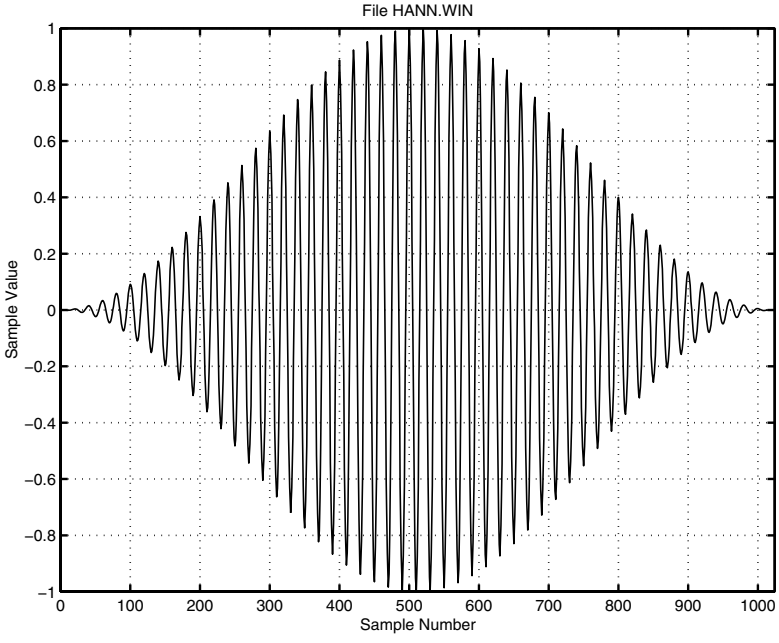


(a)

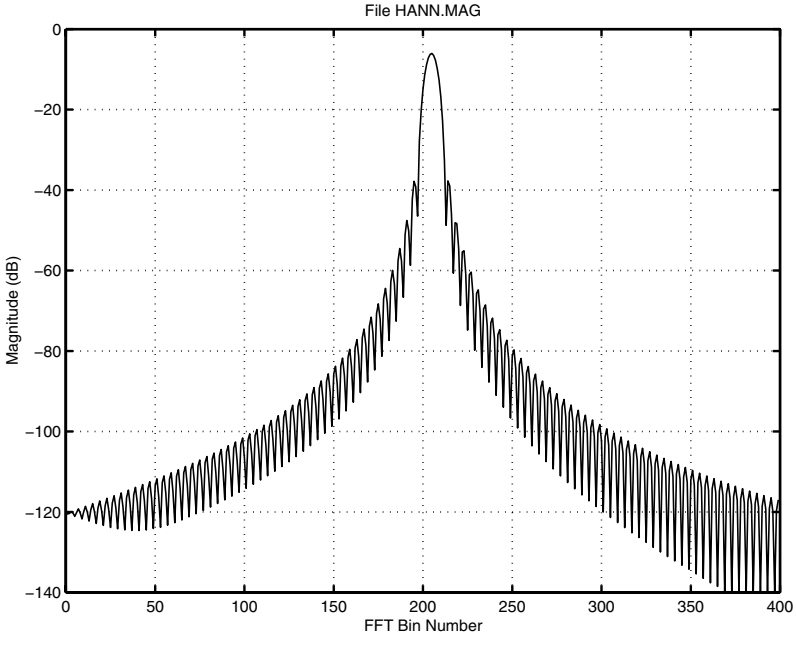


(b)

FIGURE 5.5 (a) Hamming windowed input. (b) Spectral magnitude using a hamming window.

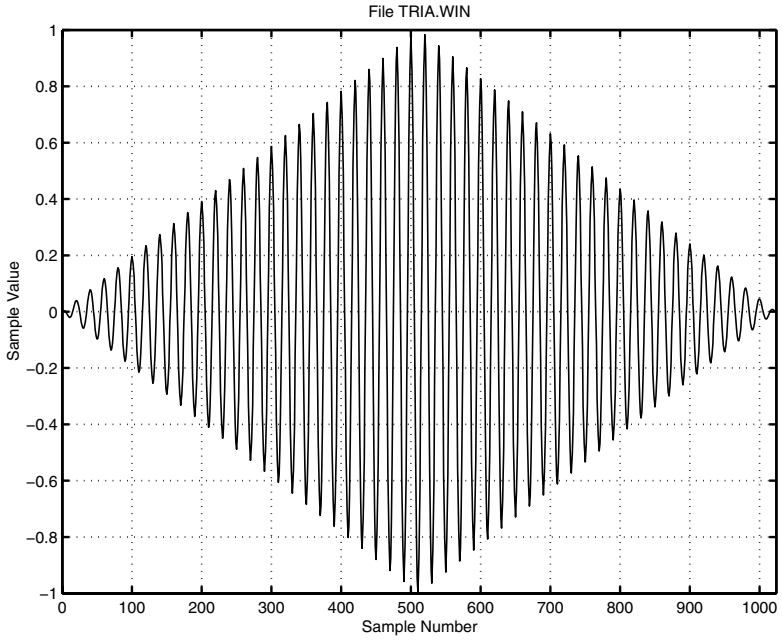


(a)

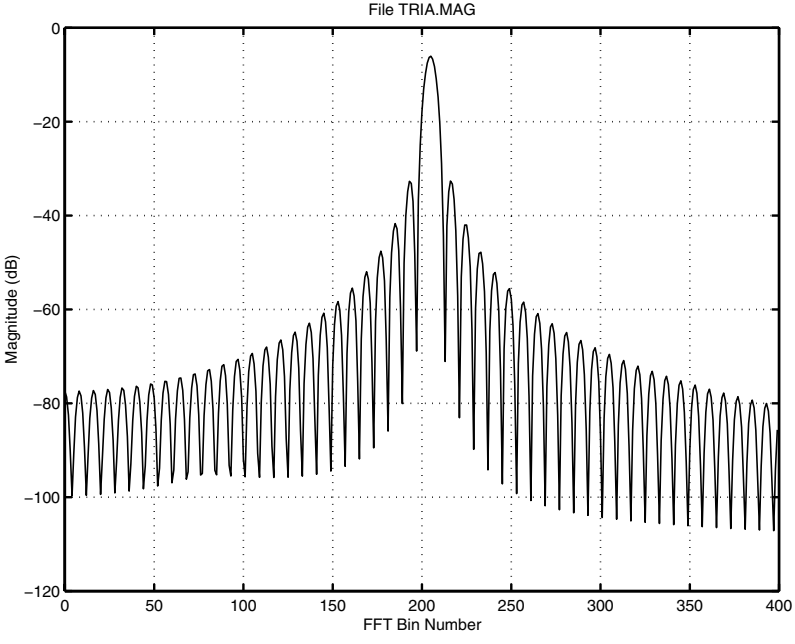


(b)

FIGURE 5.6 (a) Hanning windowed input. (b) Spectral magnitude using a Hanning window.

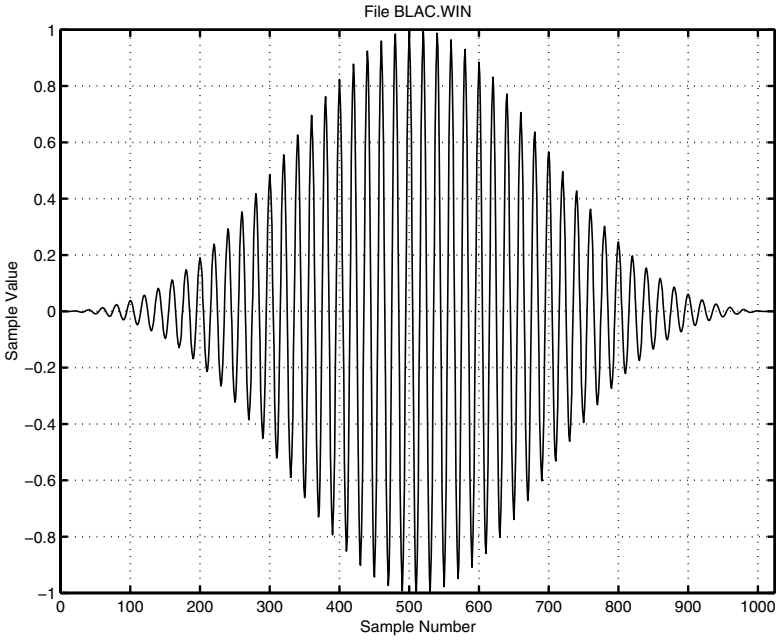


(a)

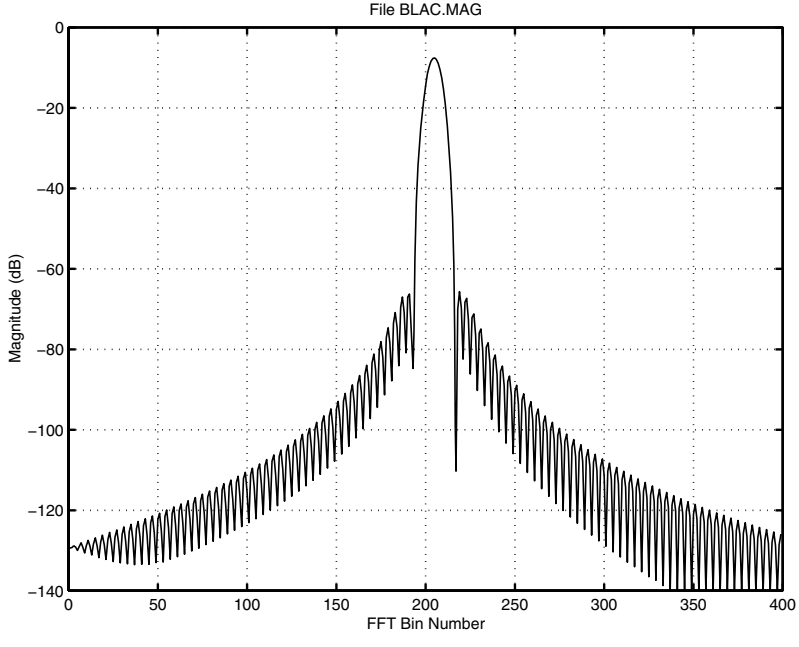


(b)

FIGURE 5.7 (a) Triangular windowed input. (b) Spectral magnitude using a triangle window.

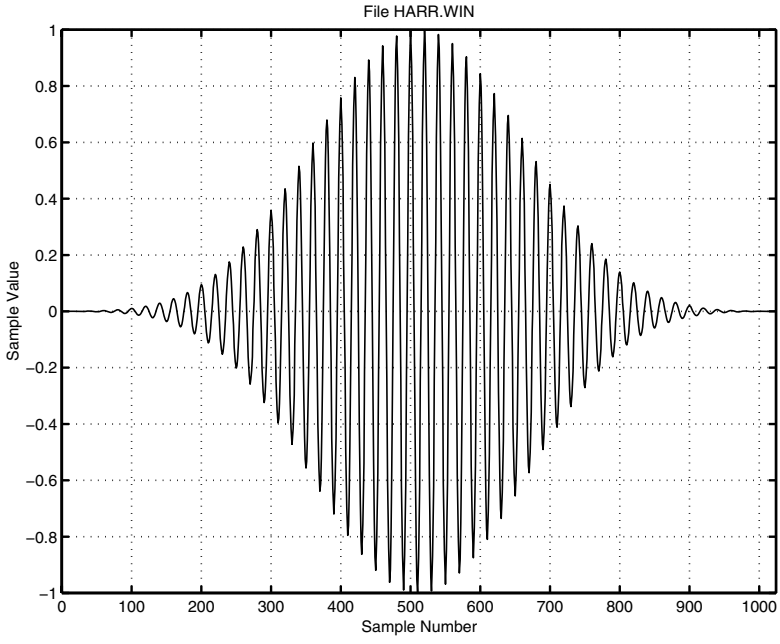


(a)

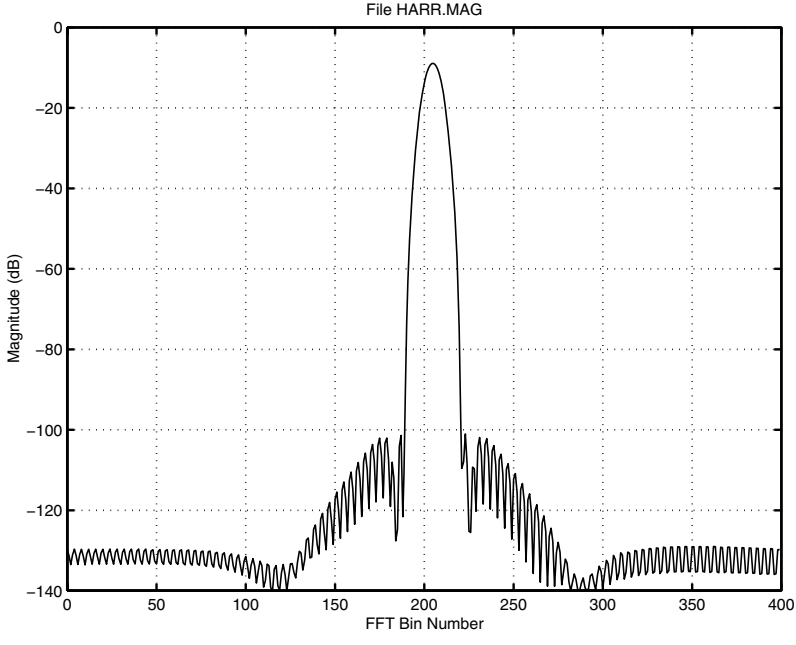


(b)

FIGURE 5.8 (a) Blackman windowed input. (b) Spectral magnitude using a Blackman window.



(a)



(b)

FIGURE 5.9 (a) Four-term Blackman-Harris widowed input. (b) Spectral magnitude using a Blackman-Harris window.

TABLE 5.1 Table of Windows and Figures of Merit (adapted from Harris).

Window	Highest Side-Lobe Level (dB)	Noise BW (BINS)	3 dB BW (BINS)	6dB BW (BINS)
Rectangle	-13	1.00	0.89	1.21
Triangle	-27	1.33	1.28	1.78
Hanning	-23	1.23	1.20	1.65
Hamming	-43	1.36	1.30	1.81
Blackman	-58	1.73	1.68	2.35
Minimum 4 point Blackman-Harris	-92	2.00	1.90	2.72

Source: Frederick J. Harris, "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform," *Proceedings of IEEE,* January 1978.

Listing 5.8. As input for this example we will create an input file to be used in the remaining examples in this chapter. It is a sum of three cosine waves at normalized frequencies (frequency divided by sample rate) of 0.01, 0.02, and 0.4. Here is the dialogue with the MKWAVE program which creates this file:

```
Enter number of samples to generate [2..10000] : 1024
Enter number of frequencies in the sum [1..20] : 3
Enter frequency #0 [0.0..0.5] : 0.01
Enter frequency #1 [0.0..0.5] : 0.02
Enter frequency #2 [0.0..0.5] : 0.4
Enter file name : WAVE1K.DAT
Signal of length 1024 equal to the sum of 3
cosine waves at the following frequencies:
f/fs = 0.01
f/fs = 0.02
f/fs = 0.4
```

```
//////////////////////////////////////////////////////////////////
//
// idfttest.cpp - Test DFT and IDFT functions
//
//////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "dft.h""
#include "disk.h"
#include "get.h"
//////////////////////////////////////////////////////////////////
```

LISTING 5.8 Program IDFTTEST used to demonstrate the DFT and inverse DFT functions. (*Continued*)

```
//
// int main()
//   Tests DFT and IDFT functions. Requires time
//   domain signal input file. Generates spectral
//   magnitude and reconstructed time domain data.
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Open the input file
        do getInput( "Enter input signal file", strName );
        while( strName.isEmpty() ||
             dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        // Read vector from file
        Vector<float> vSignal;
        dspfile.read( vSignal );
        dspfile.close();

        // Allocate vector large enough to hold data
        int lenIn = vSignal.length();
        Vector<Complex> vSamp;

        // Copy data to complex vector
        conv( vSamp, vSignal );

        // Find the spectrum of the data
        vSamp = dft( vSamp );

        // Find log magnitude and store for output
        double logMag = lenIn * lenIn;
        for( int i = 0; i < lenIn; i++ )
        {
            double tempFilt = vSamp[i].m_real * vSamp[i].m_real;
            tempFilt += vSamp[i].m_imag * vSamp[i].m_imag;
            tempFilt /= logMag;
            vSignal[i] = (float)( 10 * log10(
```

LISTING 5.8 (*Continued*)

```
                    max( tempFilt, 1.e-14 ) ) );
        }

        // Write magnitude to file
        do getInput(
         "Enter spectral magnitude file to create", strName );
        while( strName.isEmpty() );
        dspfile.openWrite( strName );
        dspfile.write( vSignal );

        // Make descriptive trailer for magnitude file
        String strTrailer;
        getInput( "Enter trailer string", strTrailer );

        if( strTrailer.isEmpty() == false )
            dspfile.setTrailer( strTrailer );
        dspfile.close();

        // Do inverse DFT
        vSamp = idft( vSamp );

        // Convert complex back to float
        for( i = 0; i < lenIn; i++ ) vSignal[i] = vSamp[i].m_real;

        // Write reconstructed signal to file
        do getInput(
         "Enter reconstructed signal file to create", strName );
        while( strName.isEmpty() );
        dspfile.openWrite( strName );
        dspfile.write( vSignal );

        // Make descriptive trailer for reconstructed signal
        getInput( "Enter trailer string", strTrailer );

        if( strTrailer.isEmpty() == false )
            dspfile.setTrailer( strTrailer );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 5.8 (*Continued*)

The example computer dialogue using the IDFTTEST program is as follows:

```
Enter the input signal file name : WAVE1K.DAT
Enter spectral magnitude file name : EXAMP2.MAG
Enter trailer string : Spectral magnitude of wave1k.dat using IDFTTEST.
Enter reconstructed signal file name : EXAMP2.REC
Enter trailer string : Reconstructed time domain file from WAVE1K.DAT.
```

Graphs of the PLOT program results of the spectral magnitude output (file EXAMP2.MAG) and the reconstructed time-domain signal (file EXAMP2.REC) are shown in Figures 5.10(a) and 5.10(b), respectively. The reconstructed output is identical to the input signal showing the **idft** is a true inverse function of **dft.**

5.8.3 Inverse FFT Test Routine

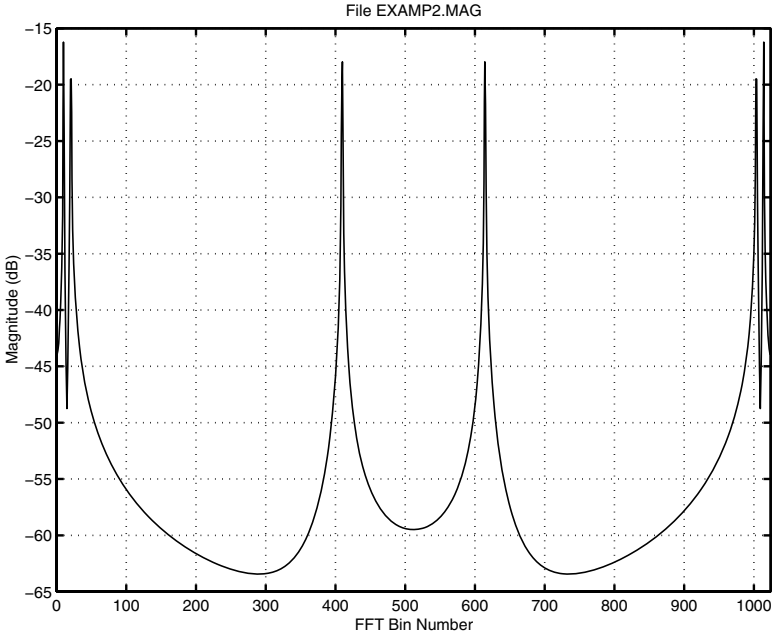
This example illustrates the use of the **fft** and **ifft** functions. The source code is shown in Listing 5.9. It accepts a time-domain input file and creates a log magnitude spectrum and a reconstructed version of the original file. The results of the IFFTTEST program are identical to the IDFTTEST program as long as the input sequence length is a power of 2. The computer dialogue using IFFTTEST is as follows:

```
Enter the input signal file name : WAVE1K.DAT
FFT size = 1024
Enter the spectral magnitude file name : ITEST.MAG
Enter the trailer string : Spectral magnitude of wave1k.dat using iffttest.
Enter reconstructed signal file name : ITEST.REC
Enter trailer string : Reconstructed time domain file from wave1k.dat.
```

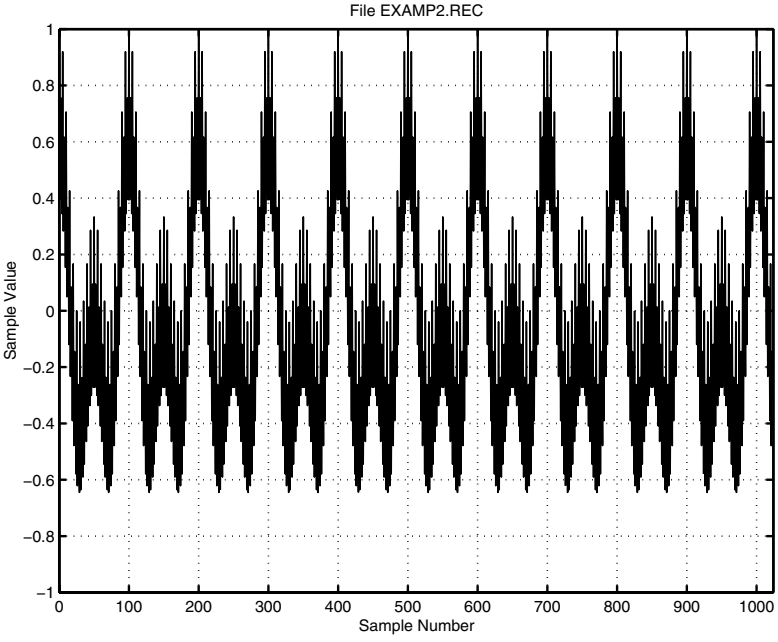
It is evident from the relative running times of the IDFTEST and IFFTTEST program that the FFT provides a very large speed-up factor when compared with the DFT. With no knowledge of the implementation of the code in each algorithm, the relative speed can be predicted using the ratio of the formulas for the number of complex multiplies for each algorithm as follows:

$$\text{speed - up factor} = \frac{N^2}{(N/2)\log_2(N)} = \frac{2N}{\log_2(N)}$$

which is a speed-up factor of 205 times for a 1024-point FFT. The actual speed-up factor is 163 times for an IBM PC. This reduced speed-up can be explained by the more time-consuming address generation in the FFT and the overhead incurred in the FFT bit reversing.



(a)



(b)

FIGURE 5.10 Results of using program IDFTTEST on file WAVE1K.DAT. (a) Spectral magnitude of WAVE1K.DAT. (b) Reconstructed waveform after a DFT and IDFT.

```
//////////////////////////////////////////////////////////////////
//
// iffttest.cpp - Test FFT and IFFT functions
//
//////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "dft.h"
#include "disk.h"
#include "get.h"
//////////////////////////////////////////////////////////////////
//
// int main()
//   Tests FFT and IFFT functions. Requires time
//   domain signal input file. Generates spectral
//   magnitude and reconstructed time domain data.
//
//  Returns:
//   0 --- Success
//
//////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Open the input file
        do getInput( "Enter input signal file", strName );
        while( strName.isEmpty() ||
             dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        // Read vector from file
        Vector<float> vSignal;
        dspfile.read( vSignal );
        dspfile.close();

        // Get length of FFT
        int lenIn = vSignal.length();
        int m = log2( lenIn );

        // Allocate vector large enough to hold data (padded to 2^n)
        int lenFFT = 1 << m;
```

LISTING 5.9 Program IFFTTEST used to demonstrate the FFT and inverse FFT functions. (*Continued*)

```
Vector<Complex> vSamp( lenFFT );
vSamp = (Complex)0.0f;

cout << "FFT size = " << lenFFT << endl;

// Copy windowed data and write it to file
for( int i = 0; i < lenIn; i++ )
    vSamp[i] = vSignal[i];

// Find the spectrum of the data
vSamp = fft( vSamp, m );

// Find log magnitude and store for output
double logMag = lenIn * lenIn;
Vector<float> logMagView( lenFFT );
for( i = 0; i < lenFFT ; i++ )
{
    double tempFilt = vSamp[i].m_real * vSamp[i].m_real;
    tempFilt += vSamp[i].m_imag * vSamp[i].m_imag;
    tempFilt /= logMag;
    logMagView[i] = (float)( 10 * log10(
                    max( tempFilt, 1.e-14 ) ) );
}

// Write magnitude to file
do getInput(
 "Enter spectral magnitude file to create", strName );
while( strName.isEmpty() );
dspfile.openWrite( strName );
dspfile.write( logMagView );

// Make descriptive trailer for magnitude file
String strTrailer;
getInput( "Enter trailer string", strTrailer );

if( strTrailer.isEmpty() == false )
    dspfile.setTrailer( strTrailer );
dspfile.close();

// Do inverse fft
vSamp = ifft( vSamp, m );

for( i = 0; i < lenIn; i++ )
    vSignal[i] = vSamp[i].m_real;
```

LISTING 5.9 (*Continued*)

```
        // Write reconstructed signal to file
        do getInput(
        "Enter reconstructed signal file to create"", strName );
        while( strName.isEmpty() );
        dspfile.openWrite( strName );
        dspfile.write( vSignal );

        // Make descriptive trailer for reconstructed signal
        getInput( "Enter trailer string", strTrailer );

        if( strTrailer.isEmpty() == false )
            dspfile.setTrailer( strTrailer );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 5.9 (*Continued*)

5.8.4 Real FFT Test Routine

This program illustrates the use of the **rfft** function. The source code is shown in Listing 5.10. It accepts an input file from disk and applies the FFT optimized for real sequences using trigonometric recombination. The computer dialogue for RFFTTEST is as follows:

```
Enter the input signal file name : WAVE1K.DAT
Enter center of magnitude file [0..511] : 256
Enter length of magnitude file [10.512] : 512
Enter spectral magnitude file name : RWAVE.MAG
Enter trailer string : spectral magnitude file from RFFTTEST
```

The results from RFFTTEST (shown in Figure 5.11) are the same as the positive frequency side of the spectral magnitude files created by IFFTTEST or IDFTTEST as shown in Figure 5.10(a).

5.9 FAST CONVOLUTION USING THE FFT

The FFT is an extremely useful tool for spectral analysis, and the programs so far in this chapter have illustrated this application. Another important application for which FFTs are often used is fast convolution. The formulas for convolution were given in Chapter 1.

```
//////////////////////////////////////////////////////////////////
//
// rffttest.cpp - Test the rfft function for real FFTs
//
//  Inputs:
//   DSPFile of time domain input signal with
//   power of 2 record length.
//
//  Outputs:
//   DSPFile with spectral magnitude.
//
//////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "dft.h"
//////////////////////////////////////////////////////////////////
//
// int main()
//
//  Returns:
//   0 -- Success
//
//////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;

        // Open input signal file
        String str;
        do getInput( "Enter input signal file name", str );
        while( str.isEmpty() || dspfile.isFound( str ) == false );

        // Read data
        Vector<float> vIn;
        dspfile.openRead( str );
        dspfile.read( vIn );
        dspfile.close();

        int len = vIn.length();

        // Perform trigonometric recombination of real input
        Vector<Complex> vOut = rfft( vIn );
        len = vIn.length();
```

LISTING 5.10 Program RFFTTEST used to demonstrate the real FFT function, **rfft**. (*Continued*)

```
        for( int i = 0; i < len / 2; i++ )
        {
            double tempflt = vOut[i].m_real * vOut[i].m_real;
            tempflt += vOut[i].m_imag * vOut[i].m_imag;
            tempflt /= ( len * len );
            vOut[i].m_real = (float)(10*log10(max(tempflt,1.e-14)));
        }

        // Get format parameters
        int center = 0;
        getInput( "Enter center of magnitude", center, 0,(len/2)-1);
        int view = 0;
        getInput( "Enter length of magnitude", view, 10, len / 2 );
        int begin = center - ( view / 2 );

        for( int k = 0; k < view; k++ )
        {
            i = k + begin;
            if( i < 0 )
                i = 0;
            else if( i >= len / 2 )
                i = ( len / 2 ) - 1;
            vIn[k] = vOut[i].m_real;
        }

        // Open output magnitude file
        do getInput(
         "Enter spectral magnitude signal file name", str );
        while( str.isEmpty() );

        dspfile.openWrite( str );
        dspfile.write( vIn( 0, view ) );

        // Get descriptive trailer
        getInput( "Enter trailer string", str );
        if( str.isEmpty() == false )
            dspfile.setTrailer( str );

        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }

    return 0;
}
```

LISTING 5.10 (*Continued*)

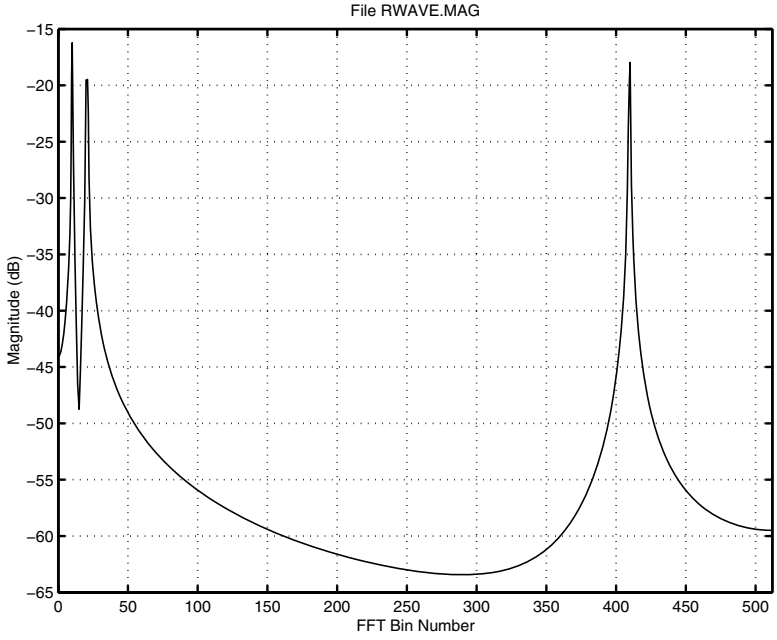


FIGURE 5.11 Spectral magnitude of file WAVE1K.DAT generated using the real-input FFT program, RFFTTEST. This graph is the same as the first half of Figure 5.10(a).

Most often a relatively short sequence, say, 20 to 200 points in length (e.g., an FIR filter), must be convolved with a number of longer input sequences. The input sequence length might be 1000 samples or greater and may be changing with time as, for instance, new data samples are taken.

One method for computation given this problem is straight implementation of the time-domain convolution equation as discussed extensively in Chapter 4. The number of real multiplies required is $M * (N - M + 1)$, where N is the input signal size and M is the length of the FIR filter to be convolved with the input signal. There is an alternative to this rather lengthy computation method that can be arrived at by using the convolution theorem. The convolution theorem states that time-domain convolution is equivalent to multiplication in the frequency domain. The convolution equation above can be rewritten in the frequency domain as follows:

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

This means that if the frequency domain representations of $h(n)$ and $x(n)$ are known, then $Y(k)$ can be calculated by simple multiplication. The sequence $y(n)$ can then be obtained by inverse Fourier transform. This sequence of steps is detailed below:

1. Create the array $H(k)$ from the impulse response $h(n)$ using the FFT.
2. Create the array $X(k)$ from the sequence $x(n)$ using the FFT.
3. Multiply H by X point by point thereby obtaining $Y(k)$.
4. Apply the inverse FFT to $Y(k)$ in order to create $y(n)$.

There are several points to note about this procedure. First, very often the impulse response, $h(n)$, of the filter does not change over many computations of the convolution equation. Therefore, the array $H(k)$ need only be computed once and can be used repeatedly, saving a large part of the computation burden of the algorithm. Second, it must be noted that $h(n)$ and $x(n)$ may have lengths different from each other, as mentioned at the beginning of this section. In this case it is necessary to create two equal-length sequences by adding zero value samples at the end of the shorter of the two sequences. This is commonly called *zero filling* or *zero padding.* This is necessary because all FFT lengths in the procedure must be equal. Also, when using the radix 2 FFT all sequences to be processed must have a power of 2 length. This can require zero filling of both sequences to bring them up to the next higher value that is a power of 2.

Finally, in order to minimize circular convolution edge effects [the distortions that occur at computation points where each value of $h(n)$ does not have a matching value in $x(n)$ for multiplication], the length of $x(n)$ is often extended by the original length of $h(n)$ by adding zero values to the end of the sequence. The problem can be visualized by thinking of the convolution equation as a process of sliding a short sequence, $h(n)$, across a longer sequence, $x(n)$, and taking the sum of products at each translation point. As this translation reaches the end of the $x(n)$ sequence, there will be sums where not all $h(n)$ values match with a corresponding $x(n)$ for multiplication. At this point the output, $y(n)$, is actually calculated using points from the beginning of $x(n)$ which may not be as useful as at the other central points in the convolution. This circular convolution effect cannot be avoided when using the FFT for fast convolution, but by zero filling the sequence its results are made more predictable and repeatable.

The speed of the FFT is what makes convolution using the Fourier transform a practical technique. In fact, in many applications, fast convolution using the FFT can be significantly faster than normal time-domain convolution. As with other FFT applications, there is less advantage with shorter sequences and with very small lengths the overhead can create a penalty. The number of real multiply/accumulate operations required for fast convolution of an N length input sequence (where N is a large number, a power of 2 and real FFTs are used) with a fixed filter sequence is $2 * N * [1 + 2*\log_2(N)]$. For example, when N is 1024 and M is 100, fast convolution is 2.15 times faster.

The program FASTCON (see Listing 5.11) illustrates the use of the **fft** and **ifft** functions for fast convolution. The convolution problem is filtering with the 35-tap lowpass FIR filter with 3 dB point at 0.2 relative frequency as was used in Chapter 4. The filter is defined in the FILTER.H header file (variable **FIRLPF35**) and the magnitude of the filter transfer function is one of the results of the program.

```
////////////////////////////////////////////////////////////////////////
//
// fastcon.cpp - Fast convolution
//
////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "dft.h"
#include "filter.h"
#include "disk.h"
#include "get.h"
#include "fircoefs.h"
////////////////////////////////////////////////////////////////////////
//
// int main()
//   Performs fast convolution using the FFT. It performs
//   the convolution require to implement a 35 point FIR filter
//   (see FIRLPF declared in fircoefs.h) on an arbitrary length
//   input file specified by the user. The filter is an LPF
//   with 40dB out of band rejection. The 3dB point is at a
//   relative frequency of approximately .25*fs.
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Open the input file
        do getInput( "Enter file to filter", strName );
        while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        // Read vector from file
        Vector<float> vSignal;
        dspfile.read( vSignal );
        dspfile.close();

        // Allocate vector large enough to hold data (padded to 2^n)
        int lenIn = vSignal.length();
```

LISTING 5.11 Program FASTCON used to demonstrate fast convolution using the **fft** and **ifft** functions. (*Continued*)

```
int m = log2( lenIn );
int lenFFT = 1 << m;
Vector<Complex> vSamp( lenFFT );
vSamp = (Complex)0.0f;

// Copy data into sample
for( int i = 0; i < lenIn; i++ )
    vSamp[i] = (Complex)vSignal[i];

// Start FFT processing
vSamp = fft( vSamp, m );

// Place the log magnitude of the FFT in a DSPFile
double tempFilt = 0.0;
float logMag = (double)lenIn * (double)lenIn;
for( i = 0; i < lenIn; i++ )
{
    tempFilt = vSamp[i].m_real * vSamp[i].m_real;
    tempFilt += vSamp[i].m_imag * vSamp[i].m_imag;
    tempFilt /= logMag;
    vSignal[i] = (float)( 10 * log10( max( tempFilt, 1.e-14 ) ) );
}

// Get spectral magnitude file
do
    getInput(
        "Enter original signal spectral magnitude file to create",
        strName );
while( strName.isEmpty() );
dspfile.openWrite( strName );
dspfile.write( vSignal );

// Make descriptive trailer for magnitude file
String strTrailer;
getInput( "Enter trailer string", strTrailer );

if( strTrailer.isEmpty() == false )
    dspfile.setTrailer( strTrailer );
dspfile.close();

// Zero fill the filter to the sequence length
Vector<Complex> vFilter( lenFFT );
vFilter = (Complex)0.0f;
int sizeFilt = ELEMENTS( FIRLPF35 );
```

LISTING 5.11 (*Continued*)

```
for( i = 0; i < sizeFilt; i++ )
    vFilter[i].m_real = FIRLPF35[i];

// FFT the zero filled filter impulse response
vFilter = fft( vFilter, m );

// Scale the filter magnitude to filter length
float scale = sizeFilt * sizeFilt;
for( i = 0; i < lenIn; i++ )
{
    tempFilt = vFilter[i].m_real * vFilter[i].m_real;
    tempFilt += vFilter[i].m_imag * vFilter[i].m_imag;
    tempFilt /= scale;
    vSignal[i] = (float)( 10 * log10( max( tempFilt, 1.e-14 ) ) );
}

// Write filter magnitude transfer function to file
do
    getInput(
        "Enter filter spectral magnitude file to create",
        strName );
while( strName.isEmpty() );
dspfile.openWrite( strName );
dspfile.write( vSignal );

// Make descriptive trailer for magnitude file
dspfile.setTrailer(
    "Spectrum of impulse response of 35 tap FIR filter.\n" );
dspfile.close();

// Multiply the two transformed sequences
for( i = 0; i < lenIn; i++ )
    vSamp[i] = vSamp[i] * vFilter[i];

// Inverse FFT the multiplied sequences
vSamp = ifft( vSamp, m );

// Write the result out to a DSPFile
for( i = 0; i < lenIn; i++ )
    vSignal[i] = vSamp[i].m_real;

// Write filter magnitude transfer function to file
do getInput( "Enter filter output signal file", strName );
while( strName.isEmpty() );
```

LISTING 5.11 (*Continued*)

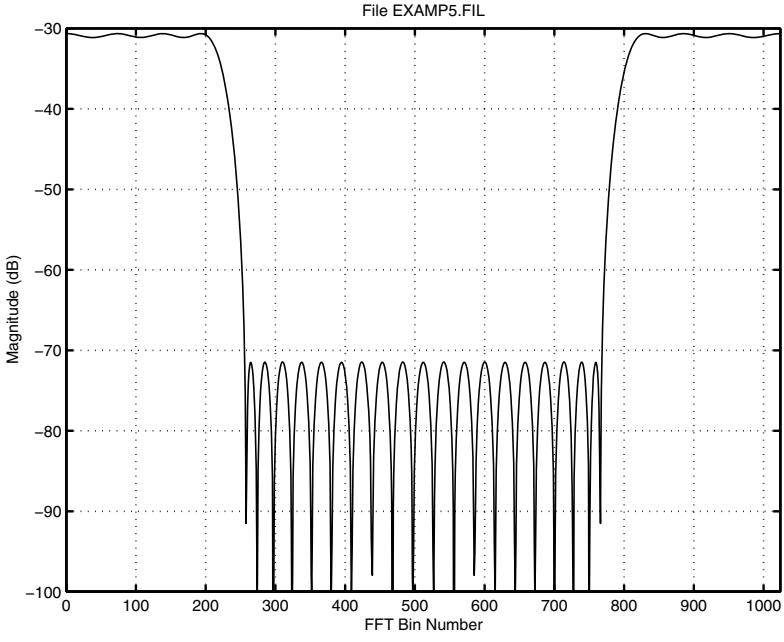
```
        dspfile.openWrite( strName );
        dspfile.write( vSignal );

        // Make descriptive trailer for reconstructed signal file
        getInput( "Enter trailer string", strTrailer );

        if( strTrailer.isEmpty() == false )
            dspfile.setTrailer( strTrailer );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 5.11 (*Continued*)

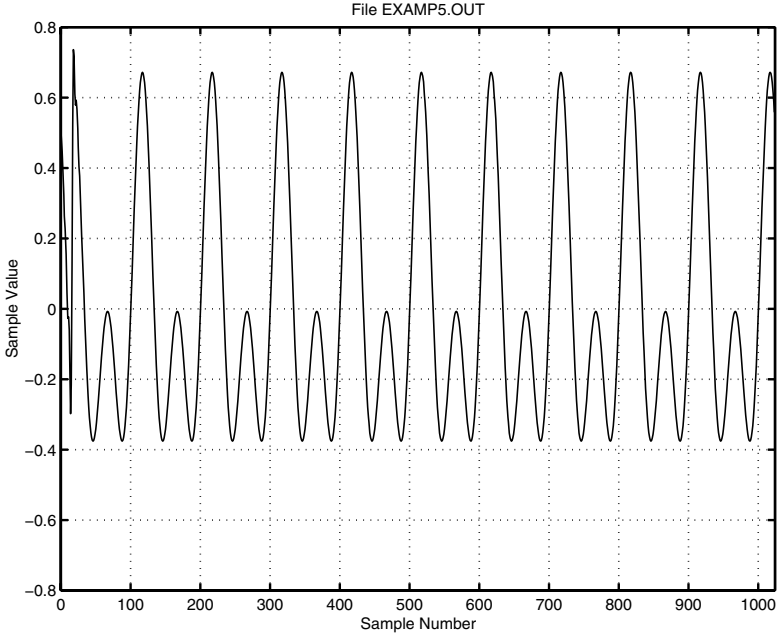
The input used in the example is the same one used for the previous examples. The files output by the program are the filter transfer function magnitude and the output time-domain response. The following illustrates the FASTCON program:

```
Enter the input signal file name : WAVE1K.DAT
Enter original signal spectral magnitude file name : EXAMP5.MAG
Enter the trailer string : FFT spectral magnitude for WAVE.DAT
Enter filter impulse response spectral magnitude file name : EXAMP5.FIL
Enter filter output signal file name : EXAMP5.OUT
Enter the trailer string: 35 tap FIR filter output from FASTCON
```

The results of the program are shown in Figure 5.12. Figure 5.12(a) shows the filter transfer function (file EXAMP5.FIL), and Figure 5.12(b) shows the output of the filter using fast convolution (file EXAMP5.OUT). The input spectral magnitude (file EXAMP5.MAG) is not shown in Figure 5.12 because it is the same as Figure 5.10(a). The frequency response of the filter shown in Figure 5.12(a) shows the 40 dB of out of band rejection and also indicates a DC response of -30 dB. This reduction in the spectral magnitude is due to the zero padding required to make the filter length and input length 1024 so that equal length FFTs are performed. If Figure 5.12(b) is compared with a plot of the input sequence [e.g., Figure 5.10(b)] it is clear that the highest of the three frequencies (0.4) has been nearly eliminated as a result of the lowpass characteristics of the filter.



(a)



(b)

FIGURE 5.12 Results of the FASTCON program using the WAVE1K.DAT file as input. A convolution with a 35-point FIR filter is illustrated. (a) Spectral magnitude transfer function of the 35-point FIR lowpass filter. (b) Time-domain output of the filter showing that the highest-frequency component ($0.4f_s$) has been removed.

5.10 POWER SPECTRAL ESTIMATION

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

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

1. Sample rate: determines maximum frequency to be estimated.
2. Length of FFT: determines the resolution (smallest frequency difference detectable).
3. Window: determines the amount of spectral leakage and affects resolution and noise floor.
4. Amount of overlap between successive spectra: determines accuracy of the estimate, directly effects computation time.
5. Number of spectra averaged: determines maximum rate of change of the spectra that will be detectable. Also directly effects the noise floor of the estimate.

One of the common application areas for power spectral estimation is speech recognition. The power spectra of a voice signal gives essential clues to the sound that was being made by the speaker. Almost all the information in voice signals is contained in frequencies below 3500 Hz. A common voice-sampling frequency that gives some margin above the Nyquist rate is 8000 Hz. The spectrum of a typical voice signal changes every 10 msec or 80 samples at 8,000 Hz. As a result, popular FFT sizes for speech research are 64 and 128 points.

Included on the disk with this book is a file called CHKL.DAT. This is the recorded voice of one of the authors (Dr. Embree) saying the words "Chicken Little." These sounds were chosen because of the range of interesting spectra that they produced. By plotting the data file using the PLOT program, the break between words can be seen and the relative volume can be inferred from the envelope of the waveform [see Chapter 4, Section 4.2.1 and Figure 4.6(a)].

The program PSE (see Listing 5.12) accepts as input a data file in the **DSPFile** format of any length (not necessarily power of 2). It prompts the user for power spectral estimation parameters such as FFT length, overlap, spectrums to average and then produces an output file consisting of a number of records determined by the input parameters. Each record is a power spectral estimate of the input file at a certain point in time. The following is the computer dialogue that can be used to create the figures given in the text:

```
This program does power spectral estimation
using parameters defined by you or a set of defaults.
The default parameters are:
          Length of each FFT snapshot:    64 points
          Number of FFTs to average:    16 FFTs
          Amount of overlap between each FFT: 60 points
Enter the input signal file name : CHKL.DAT
Would you like to use default parameters or define your own?
                1 — Default
                2 — Define your own
Enter your choice [1..2] : 1
The input file size is: 6000
The size of each estimate is: 124
The number of estimates is: 48
Enter your output file name: CHKL.OUT
************************************************************
48 power spectral estimates (dB) each of length 64.
```

```
//////////////////////////////////////////////////////////////////////
//
// pse.cpp - Power spectral estimation using the FFT
//
//////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "dft.h"
#include "disk.h"
#include "get.h"

//////////////////////////////////////////////////////////////////////
//
// Constants
//
//////////////////////////////////////////////////////////////////////

// Default parameters
const int DEF_FFT_SIZE    = 64; // Points
const int DEF_FFT_AVERAGE = 16; // FFT frames
const int DEF_FFT_OVERLAP = 60; // points

//////////////////////////////////////////////////////////////////////
//
// int main()
//   Calculates power spectral estimation on a file with
```

LISTING 5.12 Program PSE used to determine the power spectrum from an input sequence using the **fft** function. (*Continued*)

```
//   a series of records. The average power spectrum in
//   each block is determined and used to generate a series
//   of PSE records. The block size parameters can be
//   selected or a set of defaults can be used.
//
//  Note:
//  The default parameters are:
//   Length of each FFT snapshot: DEF_FFT_SIZE
//   Number of FFTs to average: DEF_FFT_AVERAGE
//   Amount of overlap between each FFT: DEF_FFT_OVERLAP
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        // Get input file
        DSPFile dspfile;
        String strName;

        // Open the input file
        do getInput( "Enter input signal file", strName );
        while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        // Read vector from file
        Vector<float> vSignal;
        dspfile.read( vSignal );
        dspfile.close();
        int lenIn = vSignal.length();

        // Display instructions and default parameters
        cout << "This program does power spectral estimation\n";
        cout << "using parameters defined by you or a set of defaults.\n\n";
        cout << "The default parameters are:\n";
        cout
            << " Length of each FFT snapshot:         "
            << DEF_FFT_SIZE
            << " points\n";
        cout
            << " Number of FFTs to average:         "
            << DEF_FFT_AVERAGE
```

LISTING 5.12 (*Continued*)

```
    << " FFT frames\n";
cout
    << " Amount of overlap between each FFT: "
    << DEF_FFT_OVERLAP
    << " points\n\n";
cout << "Use default parameters or define your own?\n";
cout << "    1 — Default\n";
cout << "    2 — Define your own\n";

// User parameters
int choice = 0;
int slice = DEF_FFT_SIZE;
int average = DEF_FFT_AVERAGE;
int overlap = DEF_FFT_OVERLAP;

getInput( "Enter your choice", choice, 1, 2 );
if( choice == 2 )
{
    // Limit FFT size if input is small
    int userLen = vSignal.length();
    if( userLen != 1 << log2( userLen ) )
        userLen = 1 << ( log2( userLen ) - 1 );

    do getInput(
            "Enter FFT length (must be power of 2)",
            slice,
            2,
            min( 1024, userLen ) );
    while( slice != 1 << log2( slice ) );
    getInput("Enter number of spectrums to average", average, 1, 256 );
    getInput("Enter number of overlap points", overlap, 0, slice - 1 );
}

// Calculate the number of samples in each estimation and
// the number of estimations in the data set
int estSize = ( slice - overlap ) * ( average - 1 ) + slice;

cout << "The input file size is: " << lenIn << endl;
cout << "The size of each estimate is: " << estSize << endl;

int numests = lenIn / estSize;

cout << "The number of estimates is: " << numests << endl;

Vector<float> vMag( slice );
```

LISTING 5.12 (*Continued*)

```
Vector<Complex> vSamp( slice );
vMag = 0.0f;
vSamp = (Complex)0.0f;

// Open output file
do getInput( "Enter output file name", strName );
while( strName.isEmpty() );
dspfile.openWrite( strName );

// Write out output one record at a time
int k = 0;

for( int i = 0; i < numests; i++ )
{
    vMag = 0.0f;
    for( int j = 0; j < average; j++ )
    {
        for( k = 0; k < slice; k++ )
        {
            int index = i * estSize + j * ( slice - overlap ) + k;
            vSamp[k].m_real = vSignal[index];
            vSamp[k].m_imag = 0.0f;
        }

        // Hamming window over samples
        vSamp = ham( vSamp );

        int m = log2( slice );
        vSamp = fft( vSamp, m );

        float a = slice * slice;
        for( k = 0; k < slice; k++ )
        {
            double tempflt = vSamp[k].m_real * vSamp[k].m_real;
            tempflt += vSamp[k].m_imag * vSamp[k].m_imag;
            tempflt = tempflt / a;
            vMag[k] = (float)( vMag[k] + tempflt );
        }
    }

    // Take log after averaging the magnitudes
    for( k = 0; k < slice / 2; k++ )
    {
        vMag[k] = vMag[k] / average;
```

LISTING 5.12 (*Continued*)

```
                vMag[k] = (float)( 10 * log10( max( vMag[k], 1.e-14f ) ) );
            }
            dspfile.write( vMag( 0, slice / 2 ) );
        }
        // Make descriptive trailer for magnitude file
        char szTrailer[300];
        sprintf(
            szTrailer,
            "\n%d power spectral estimates (dB) each from FFT length = %d.\n",
            numests,
            slice );

        cout << szTrailer;

        dspfile.setTrailer( szTrailer );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
        return 0;
}
```

LISTING 5.12 (*Continued*)

Two of the spectra from the output file of this example are shown in Figure 5.13. If the reader runs the example as shown above, all the snapshots can be plotted and the course of the changing voice spectra can be traced. It is also interesting to experiment with different parameters for length, overlap, and number to average. Figure 5.14 shows an image of the resulting spectra plotted as a frequency versus time plot with the amplitude of the spectrum indicated by the darkness of each pixel. The high-frequency content of the "chi" part of chicken and the lower frequency content of "little" are clearly indicated.

5.11 INTERPOLATION USING THE FOURIER TRANSFORM

In Chapter 4, Section 4.5, time-domain interpolation was discussed and demonstrated using several short FIR filters. In this section, the same process is demonstrated using FFT techniques. The steps involved in 2:1 interpolation using the FFT are as follows:

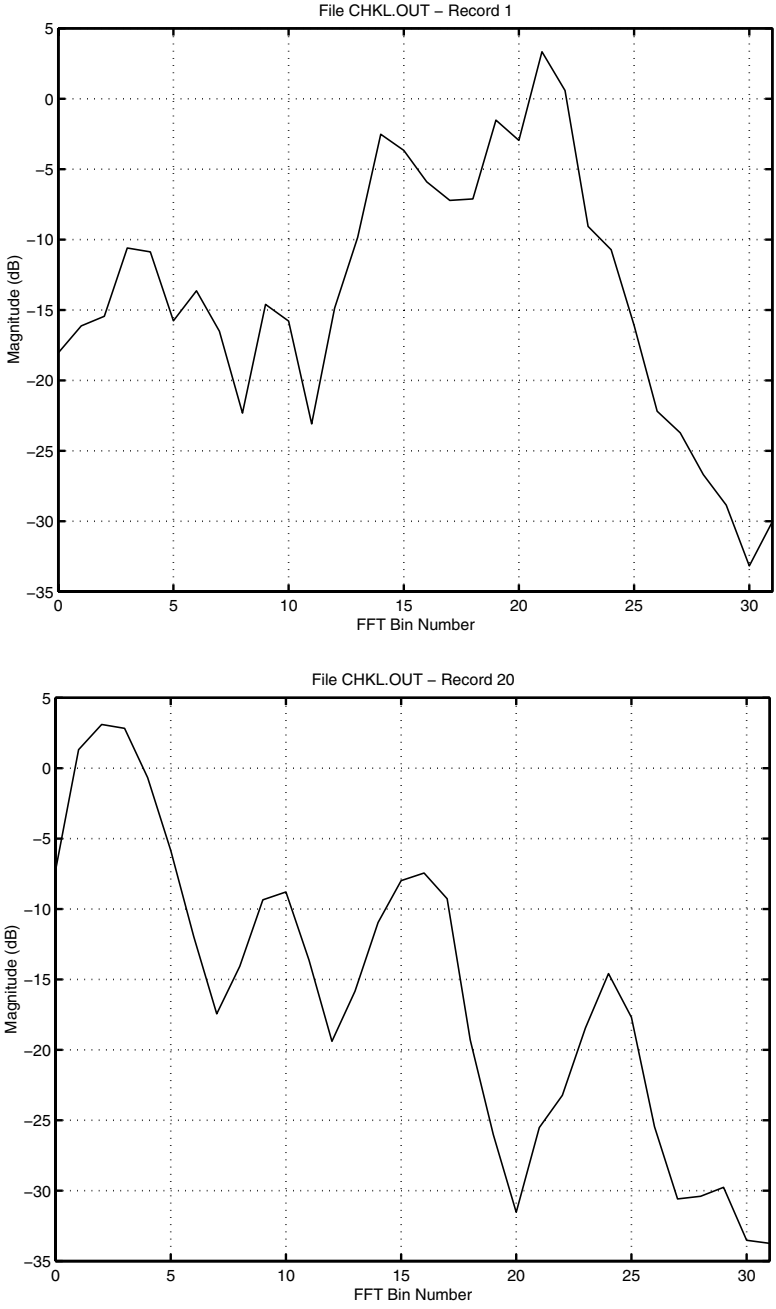


FIGURE 5.13 Results of the program PSE, which performs power spectral estimation using the FFT. The input is the speech signal file CHKL.DAT [see Figure 4.6(a)]. (a) High-frequency character of the "ch" sound in the first word "chicken" (record 1). (b) Lower-frequency content of the "l" sound at the beginning of the second word "little" (record 20).

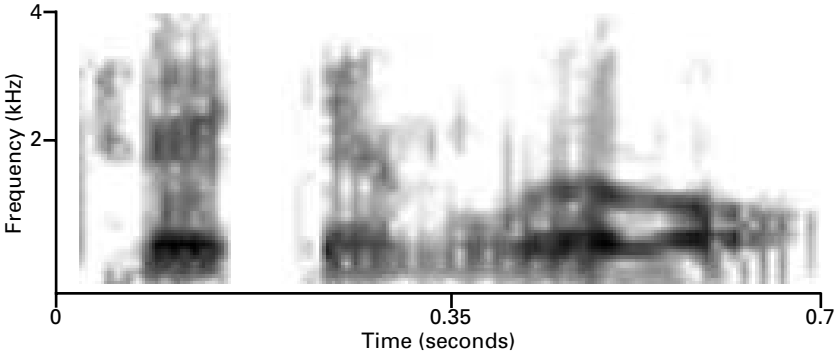


FIGURE 5.14 Image of the power spectrum versus frequency and time obtained using the PSE program with the input file CHKL.DAT. Dark areas indicate a large-magnitude frequency component.

1. Perform an FFT with a power of 2 length (N) that is greater than or equal to the length of the input sequence.
2. Zero pad the frequency domain representation of the signal (a complex array) by inserting N - 1 zeroes between the positive and negative half of the spectrum. The Nyquist frequency sample output of the FFT (at the index $N/2$) is divided by 2 and placed with the positive and negative parts of the spectrum (this results in a symmetrical spectrum for a real input signal).
3. Perform an inverse FFT with a length of $2N$.
4. Multiply the interpolated result by a factor of 2 and copy the desired portion of the result that represents the interpolated input (this is all the inverse FFT samples if the input length was a power of 2).

Listing 5.13 shows the program INTFFT, which performs 2:1 interpolation using the above procedure and the **fft** and **ifft** functions. Figure 5.15(b) shows the result of using the INTFFT program on the first 128 samples of the WAVE.DAT input file used in the previous examples in this chapter [these 128 samples are shown in detail in Figure 5.15(a)]. Note that the length is twice as large (256) and more of the cosine nature of the waveform can be seen in the interpolated result. The INTFFT program can be used to interpolate any signal by a power of 2 by using it repeatedly until the desired interpolation factor is reached. Also, because the FFT is employed, frequencies as high as the Nyquist rate can be accurately interpolated. FIR filter interpolation has a upper frequency limit because of the frequency response of the filter (see Section 4.5 of Chapter 4).

The examples presented in this chapter provide several prototypes for the Fourier transform and utility code introduced here. By copying the use of the functions the reader can implement custom code using **fft**, **rfft**, **dft**, and window functions. The last few sections have also provided many examples of the practice use of

```
////////////////////////////////////////////////////////////////////////
//
// intfft.cpp - Interpolate 2:1 using FFT and IFFT functions
//
////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "dft.h"
#include "disk.h"
#include "get.h"
////////////////////////////////////////////////////////////////////////
///
//
// int main()
//   Accepts time domain signal input file.
//   Generates 2:1 interpolated time domain data.
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////////////
///
int main()
{
    try
    {
        DSPFile dspfileIn,dspfileOut;
        String str;

        // Open the input file
        do getInput( "Enter input signal file", str );
        while( str.isEmpty() || dspfileIn.isFound( str ) == false );
        dspfileIn.openRead( str );

        // Get length of FFT
        int lenIn = dspfileIn.getRecLen();
        int m = log2( lenIn );
        int lenFFT = 1 << m;
        if( lenIn != lenFFT )
        {
            cout
                << "Warning: Zero-padding signal to "
                << ( 1 << m ) << " samples for FFT\n";
        }

        cout << "FFT size = " << lenFFT << endl;
```

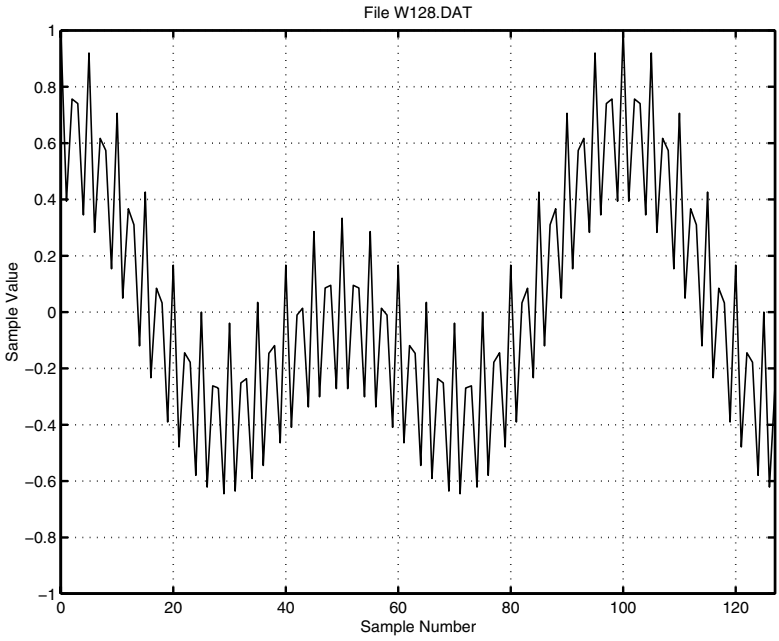
LISTING 5.13 Program INTFFT used to interpolate a signal by a 2:1 ratio using the **fft** and **ifft** functions. (*Continued*)

```
        do getInput( "Enter output file name", str );
        while( str.isEmpty() );
        dspfileOut.openWrite( str );

        // Read in file record by record
        for( int j = 0; j < dspfileIn.getNumRecords(); j++ )
        {

            // Read vector from file
            Vector<float> vSignal;
            dspfileIn.read( vSignal );

            // Perform FFT on complex
            Vector<Complex> vSamp( 2 * lenFFT );
            vSamp = (Complex)0.0f;
            for( int i = 0; i < lenIn; i++ )
                vSamp[i].m_real = vSignal[i];

            vSamp = fft( vSamp, m );

            // Divide the middle frequency component by 2
            vSamp[lenFFT / 2].m_real /= 2;
            vSamp[lenFFT / 2].m_imag /= 2;

            // Zero pad and move the negative frequencies
            vSamp[3 * lenFFT / 2] = vSamp[lenFFT / 2];
            for( i = lenFFT / 2 + 1; i < lenFFT; i++ )
            {
                vSamp[i + lenFFT] = vSamp[i];
                vSamp[i].m_real = 0.0F;
                vSamp[i].m_imag = 0.0F;
            }

            // Do inverse FFT
            vSamp = ifft( vSamp, m + 1 );

            // Create real output samples
            Vector<float> vOut( 2 * lenIn );

            // Copy real part to output and multiply by 2
            for( i = 0; i < 2 * lenIn; i++ )
                vOut[i] = 2.0 * vSamp[i].m_real;

            // Write magnitude to file
            dspfileOut.write( vOut );
```

LISTING 5.13 (*Continued*)

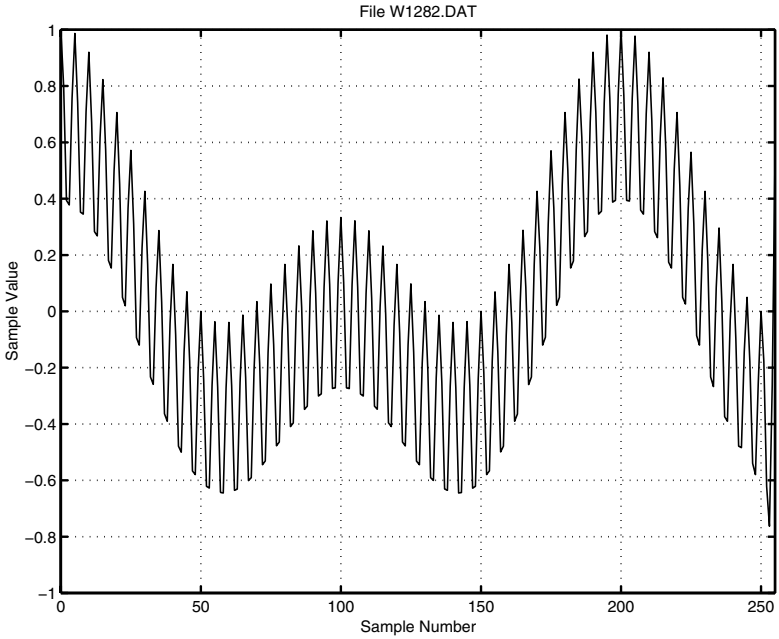
```
        }
        // Make descriptive trailer for reconstructed signal file
        getInput( "Enter trailer string", str );

        if( str.isEmpty() == false )
            dspfileOut.setTrailer( str );
        dspfileIn.close();
        dspfileOut.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 5.13 (*Continued*)



(a)

FIGURE 5.15 Example use of the INTFFT program used to interpolate a signal by a 2:1 ratio. (a) 128-sample input signal (the beginning of WAVE.DAT). (b) 256-sample interpolated result.



(b)

FIGURE 5.15 (*Continued*)

the FFT and DFT. In addition to serving as good example code, the programs can be useful in themselves for spectrum analysis of input files in the **DSPFile** format. The programs FFTTEST, PSE, and INTFFT are particularly usable in their current form.

5.12 EXERCISES

1. Run MKWAVE to generate file FREQ05.DAT with a single frequency at 0.05 with 1024 points generated. Run FFTTEST using FREQ05.DAT with a selected window. The power of 2 length of the FFT should be 12 (FFT length of 4096). Use a center of magnitude of 200 (the FFT bin number that will appear at the center of the magnitude output record) and a length of the magnitude record of 400. Plot the magnitude file to compare different windows.
2. Run FFTTEST using the data file generated in Exercise 1 with a power of 2 length of 10 instead of 12. Why are the results different?
3. Run MKWAVE again with 1000 points instead of 1024. Run FFTTEST again with the same window. Why is the magnitude output different? Also try a 1000-point DFT using the program DFTTEST without windowing to see spectral changes due to this relatively small record length change.
4. Run MKWAVE again with 1024 points and a frequency of 0.0625. Run FFTTEST again with the same window and a power of 2 FFT length of 10. Why is the magnitude output different?

5. Run MKWAVE to generate file FREQ2.DAT with a two frequencies at 0.05 and 0.07 with 1024 points generated. Run FFTTEST using FREQ2.DAT with a selected window. The power of 2 length of the FFT should be 12 (FFT length of 4096). Use a center of magnitude of 200 (the FFT bin number that will appear at the center of the magnitude output record) and a length of the magnitude record of 400. Plot the magnitude file to compare different windows. Which windows can separate the two frequencies?
6. Run the FFTTEST program using the CHKL.DAT speech file as input using your favorite window. Use a center of magnitude of 1024 and a length of 2048 to display the full frequency spectrum. What does the output magnitude file show about the speech data?
7. Run PSE using a signal containing two cosine waves generated using MKWAVE as input. Observe the simple spectrum. Try different PSE parameters and observe the results. How long does the FFT length need to be to see the two cosine waves seperated by a frequency of 0.1?
8. Run PSE using the chirp signal generated by the REALTIME program as input. Observe the spectrum change with time. Try different PSE parameters and observe the results. Try different windows by recompiling the PSE program. Who does the output spectrum change? How long does the FFT length need to be to see the slow changes in the frequency of the chirp?
9. Run FASTCON using WAVE3.DAT as input (a signal generated using MKWAVE with frequencies 0.01, 0.02, and 0.4). Compare the results of this filtering program with the FIR lowpass filter output obtained using the FIRFILT program (as was illustrated in Chapter 4). Magnitude files are also generated by FASTCON in order to observe the spectrum of the input signal and the FIR filter frequency response.
10. Run INTFFT using WAVE3.DAT (see Exercise 9) as the input file. Plot the 2:1 interpolated output and compare it to the result obtained using the FIR interpolation techniques discussed in Chapter 4.

5.13 REFERENCES

BRIGHAM, E. O., *The Fast Fourier Transform.* Prentice Hall, Englewood Cliffs, NJ, 1974.

BRIGHAM, E. O., *The Fast Fourier Transform and Its Applications,* Prentice Hall, Englewood Cliffs, NJ, 1988.

BRISTOW, G., ed., *Electronic Speech Recognition,* McGraw-Hill, New York, 1986.

BURRUS, C. S., and PARKS, T. W., *DFT/FFT and Convolution Algorithms,* John Wiley and Sons, New York, 1985.

ELLIOTT, D. F., ed., *Handbook of Digital Signal Processing,* Academic Press, San Diego, CA, 1987.

HARRIS, FREDRIC J., "On the Use of Windows for Harmonic Analysis With the Discrete Fourier Transform," *Proceedings of IEEE,* January 1978.

MARPLE, S. L., *Digital Spectral Analysis With Applications,* Prentice Hall, Englewood Cliffs, NJ, 1987.

PAPOULIS, A., *The Fourier Integral and Its Applications,* McGraw-Hill, New York, 1962.

RABINER, L. R., and GOLD, B., *Theory and Application of Digital Signal Processing,* Prentice Hall, Englewood Cliffs, NJ, 1975.

CHAPTER 6

Matrix and Vector Routines

This chapter discusses the basic implementation of matrix and vector operations in C++. The majority of the routines are general purpose and could be used for any type of data analysis or data processing as well as digital signal processing. Although the routines presented provide a fairly complete set of the commonly used matrix and vector operations for real numbers, they are by no means exhaustive. Readers wishing to develop programs which can do more advanced matrix operations such as singular value decomposition or eigen analysis are referred to the references (e.g., *Numerical Recipes in C* by Press et al.). In particular, more advanced topics in digital spectral analysis using matrices is covered in detail in the text by Marple.

6.1 VECTOR OPERATIONS

Vector operations are one-dimensional operations that take two equal size arrays of data, perform some arithmetic operation on the arrays, and give a result. The result could be another vector, or it could be a single scalar value. For example, a vector dot product gives a scalar result equal to the sum of the products of the vector elements, while a vector product gives a vector equal to the product of each pair of elements. In fact, the tools provided by all C++ compilers are sufficient to perform all the vector operations. For example, the dot product of two arrays **a** and **b** can be found using the following code:

```
dotp = 0.0;
for(i = 0 ; i < length ; i++)
    dotp += a[i] * b[i];
```

This works as planned as long as **i**, **length**, **dotp**, and the arrays are defined (or allocated). Unfortunately, the above code is not very efficient (because of the index arithmetic), and it is tedious to write these three lines of code (and define the variables) each time a dot product is required. Perhaps the following function, which performs the dot product and returns the result, is a better implementation:

```
float dotp(a,b,length)
     float *a,*b;
     int length;
{
     float result = 0.0;
     while(length—)
       result += (*a++) * (*b++);
     return(result);
}
```

This function has the advantage that it is efficient (because the result is calculated using pointers), and it is easy to use. The problem with using functions for vector operations is that each function requires a particular type of array (in this case float arrays). Thus, if a programmer wanted to implement a dot product with different types of numbers, then a different function would be required for each combination of the **char**, **int**, **float**, and **double** data types. This would require at least 16 functions for each type of vector operation. Although this works fine, it is difficult to use because 16 unique names must be created and remembered for each vector operation. This disadvantage of functions in C is solved in the C++ object-oriented programming language. In C++, a template with a member function can be used so that the same function name can be used with different types of parameters. The compiler determines the function and operator types to use based on the type of parameters that are used in the function invocation.

6.1.1 Vector Arithmetic

Fourteen **Vector** member functions are defined in the file VECTOR.H. These functions define the following vector operations:

Operator	*Member Function Purpose*
=	Set vector equal to another vector or set to a constant (2 different member functions)
[ ]	Return an element of a vector and check index boundaries
(b,len)	Extract subvector starting at **b** with length **len**
add	Add two vectors element by element
sub	Subtract two vectors element by element
pwisemult	Multiply two vectors element by element
mult	Determine vector dot product (return scalar)

sum	Add all the elements of a vector (return scalar)
scale	Multiply all elements in a vector by a float scalar
conv	Convert one type of vector to another
<<	Display a vector to a stream
!=	Compare two vectors element by element
==	Compare two vectors element by element

The templates and definitions for all member functions in the **Vector** class are shown in Listing 6.1.

```
// Declarations for friend classes
class DSPFile;
//
// Class Vector
//  Template class for arrays
//
template <class Type>
class Vector
{
private:
    // Data contained in class Vector
    Type *m_data;

    // Length of the vector
    int m_length;
    ////////////////////////////////////////////////////////////////////////
    //
    // init( int newLen, const Type *newData = NULL )
    //   Sets length of vector to newLen and copies data newData
    //   into vector.
    //
    //  Throws:
    //   DSPException
    //
    ////////////////////////////////////////////////////////////////////////
    void init( int newLen, const Type *newData = NULL )
    {
        empty();

        // Parameter validation
        if( newLen < 0 )
            throw DSPParamException( "Vector length less than zero" );
```

LISTING 6.1 File VECTOR.H containing all **Vector** member functions. (*Continued*)

```
        // Check if zero length vector
        if( newLen == 0 )
        {
            if( newData != NULL )
              throw DSPParamException( "Vector has data but no length" ;
            return;
        }

        // Allocate new array
        m_data = new Type[ newLen ];
        if( m_data == NULL )
            throw DSPMemoryException( "Vector allocation failed" );
        m_length = newLen;

        // Check if new data was given
        if( newData != NULL )
        {
            // Copy in new data from initialization
            memcpy( m_data, newData, newLen * sizeof( Type ) );
        }
    }
public:
    ////////////////////////////////////////////////////////////////////
    //
    // Constructors and Destructor
    //
    ////////////////////////////////////////////////////////////////////
    Vector( int length = 0, const Type *newData = NULL ) :
        m_data( NULL ),
        m_length( 0 )
    {
        init( length, newData );
    }

    // Copy constructor
    Vector( const Vector<Type>& v ) :
        m_data( NULL ),
        m_length( 0 )
    {
        // Construction from existing vector
        init( v.m_length, v.m_data );
    }
    ~Vector() { empty(); }
```

LISTING 6.1 (*Continued*)

```
//////////////////////////////////////////////////////////////////////
//
// Get and Set vector information
//
//////////////////////////////////////////////////////////////////////
// Set length of vector
Vector<Type>& setLength( int newLen )
{
    init( newLen );
    return *this;
}

// Return true if vector empty
bool isEmpty() const { return ( m_data == NULL ); }

// Clear data members
void empty()
{
    delete [] m_data;
    m_data = NULL;
    m_length = 0;
}

// Return the number of elements
int length() const { return m_length; }
//////////////////////////////////////////////////////////////////////
//
// Get and Set vector contents
//
//////////////////////////////////////////////////////////////////////
// Set vector equal to vector v
Vector<Type>& operator=( const Vector<Type>& v )
{
    // Equal to itself
    if( &v != this )
        init( v.m_length, v.m_data );

    return *this;
}

// Set all elements of vector equal to constant
Vector<Type>& operator=( const Type t )
{
    Type *pt = m_data;
```

LISTING 6.1 (*Continued*)

```
    for( int i = 0; i < m_length; i++ )
        *pt++ = t;
    return *this;
}

// Return an element of the vector
Type& operator[]( int element )
{
    // Parameter validation
    if( isEmpty() )
        throw DSPBoundaryException( "Vector has no data" );

    if( element < 0 || element >= m_length )
        throw DSPBoundaryException( "Vector index out of range" );

    return m_data[element];
}

const Type operator[]( int element ) const
{
    // Class validation
    if( isEmpty() )
        throw DSPBoundaryException( "Vector has no data" );

    // Parameter validation
    if( element < 0 || element >= m_length )
        throw DSPBoundaryException( "Vector index out of range" );

    return m_data[element];
}

// Allow fast read-only access to data
const Type *getData( int len, int b = 0 ) const
{
    // Parameter validation
    if( isEmpty() )
        throw DSPBoundaryException( "Vector has no data" );

    if( b < 0 || len < 0 || b + len > m_length )
      throw DSPBoundaryException( "Data pointer indices out of range" );

    return &m_data[b];
}

// Return a new read-only subvector of this vector
const Vector<Type> operator()( int b, int len ) const
{
```

LISTING 6.1 (*Continued*)

```
        // Parameter validation
        if( isEmpty() )
            throw DSPBoundaryException( "Vector has no data" );

        if( b < 0 || len < 0 || b + len > m_length )
            throw DSPBoundaryException( "Data pointer indices out of range" );

        Vector<Type> vRet( len, &m_data[b] );
        return vRet;
    }
    //
    // Friends
    //
    // DSPFile reads and writes m_data for fast disk access
    friend class DSPFile;
};
////////////////////////////////////////////////////////////////////////
//
// add( const Vector<Type>& v1, const Vector<Type>& v2 )
//   Add two vectors.
//
//  Returns:
//   Resultant vector (v1 + v2)
//
////////////////////////////////////////////////////////////////////////
template <class Type>
Vector<Type> add( const Vector<Type>& v1, const Vector<Type>& v2 )
{
    int len = min( v1.length(), v2.length() );
    if( len == 0 )
        throw DSPBoundaryException( "Vector has no data" );
    Vector<Type> vRet( len );
    for( int i = 0; i < len; i++ )
        vRet[i] = v1[i] + v2[i];
    return vRet;
}
////////////////////////////////////////////////////////////////////////
//
// sub( const Vector<Type>& v1, const Vector<Type>& v2 )
//   Subtracts two vectors.
//
//  Returns:
//   Resultant vector (v1 - v2)
//
////////////////////////////////////////////////////////////////////////
template <class Type>
```

LISTING 6.1 (*Continued*)

```
Vector<Type> sub( const Vector<Type>& v1, const Vector<Type>& v2 )
{
    int len = min( v1.length(), v2.length() );
    if( len == 0 )
        throw DSPBoundaryException( "Vector has no data" );
    Vector<Type> vRet( len );
    for( int i = 0; i < len; i++ )
        vRet[i] = v1[i] - v2[i];
    return vRet;
}
////////////////////////////////////////////////////////////////////////
//
// pwisemult( const Vector<Type>& v1, const Vector<Type>& v2 )
//   Multiplies two vectors.
//
//  Returns:
//   Resultant vector (v1 * v2)
//
////////////////////////////////////////////////////////////////////////
template <class Type>
Vector<Type> pwisemult( const Vector<Type>& v1, const Vector<Type>& v2 )
{
    int len = min( v1.length(), v2.length() );
    if( len == 0 )
        throw DSPBoundaryException( "Vector has no data" );
    Vector<Type> vRet( len );
    for( int i = 0; i < len; i++ )
        vRet[i] = v1[i] * v2[i];
    return vRet;
}
////////////////////////////////////////////////////////////////////////
//
// mult( const Vector<Type>& v1, const Vector<Type>& v2 )
//   Calculates dot product of two vectors.
//
//  Returns:
//   Resultant scalar dot product (v1 DOT v2)
//
////////////////////////////////////////////////////////////////////////
template <class Type>
Type mult( const Vector<Type>& v1, const Vector<Type>& v2 )
{
    int len = min( v1.length(), v2.length() );
    if( len == 0 )
        throw DSPBoundaryException( "Vector has no data" );
    Type t = (Type)0;
```

LISTING 6.1 (*Continued*)

```
    for( int i = 0; i < len; i++ )
        t += v1[i] * v2[i];
    return t;
}
////////////////////////////////////////////////////////////////////////
//
// sum( const Vector<Type>& v )
//   Adds all of the elements of a vector.
//
//  Returns:
//   Resultant scalar sum
//
////////////////////////////////////////////////////////////////////////
template <class Type>
Type sum( const Vector<Type>& v )
{
    int len = v.length();
    if( len == 0 )
        throw DSPBoundaryException( "Vector has no data" );
    Type t = (Type)0;
    for( int i = 0; i < len; i++ )
        t += v[i];
    return t;
}
////////////////////////////////////////////////////////////////////////
//
// scale( float scale, const Vector<Type>& v )
//   Scales all elements of a vector by a float value.
//
//  Returns:
//   Resultant vector (s * v)
//
////////////////////////////////////////////////////////////////////////
template <class Type>
Vector<Type> scale( float scale, const Vector<Type>& v )
{
    int len = v.length();
    if( len == 0 )
        throw DSPBoundaryException( "Vector has no data" );
    Vector<Type> vRet( len );
    for( int i = 0; i < len; i++ )
        vRet[i] = (Type)(scale * v[i]);
    return vRet;
}
////////////////////////////////////////////////////////////////////////
```

LISTING 6.1 (*Continued*)

```
//
// conv( Vector<TypeTo>& vTo, const Vector<TypeFrom>& vFrom )
//   Converts a vector from one type to another.
//
// Returns:
//  Resultant vector ( vTo = (vTo's type)vFrom )
//
////////////////////////////////////////////////////////////////////////
template <class TypeTo, class TypeFrom>
Vector<TypeTo>& conv( Vector<TypeTo>& vTo, const Vector<TypeFrom>& vFrom )
{
    // Cast to void so different types can be compared
    if( (void *)&vTo == (void *)&vFrom )
        return vTo;

    int len = vFrom.length();
    if( len == 0 )
        throw DSPBoundaryException( "Vector has no data" );

    // Convert data element-by-element
    vTo.setLength( len );
    for( int i = 0; i < len; i++ )
        vTo[i] = (TypeTo)vFrom[i];

    return vTo;
}
////////////////////////////////////////////////////////////////////////
//
// operator<<( ostream& os, Vector<Type>& v )
//   Writes the vector out to an ostream for display.
//
//  Returns:
//   ostream after write
//
////////////////////////////////////////////////////////////////////////
template <class Type>
ostream& operator<<( ostream& os, const Vector<Type>& v )
{
    int len = v.length();
    os << "Vector<" << typeid( Type ).name();
    os << ">( " << len << " )\n{";
    if( v.isEmpty() )
        os << "\n};\n";
    else
    {
```

LISTING 6.1 (*Continued*)

```
        os << "\n\t";
        for( int i = 0; i < len - 1; i++ )
        {
            os << v[i] << ", ";
        }
        os << v[i] << "\n};\n";
    }
    return os;
}
////////////////////////////////////////////////////////////////////////
//
// bool operator!=( const Vector<Type>& v1, const Vector<Type>& v2 )
//   Compare two vectors element by element.
//
//  Throws:
//   DSPException
//
//  Returns:
//   true -- Vectors are different
//
////////////////////////////////////////////////////////////////////////
template <class Type>
bool operator!=( const Vector<Type>& v1, const Vector<Type>& v2 )
{
    if( v1.isEmpty() || v2.isEmpty() )
        throw DSPMathException( "Comparing empty vector" );
    if( &v1 == &v2 )
        return false;
    if( v1.length() != v2.length() )
        return true;
    for( int i = 0; i < v1.length(); i++ )
    {
        if( v1[i] != v2[i] )
            return true;
    }
    return false;
}
////////////////////////////////////////////////////////////////////////
//
// bool operator==( const Vector<Type>& v1, const Vector<Type>& v2 )
//   Compare two vectors element by element.
//
//  Throws:
//   DSPException
```

LISTING 6.1 (*Continued*)

```
//
//  Returns:
//   true -- Vectors are same
//
////////////////////////////////////////////////////////////////////
template <class Type>
bool operator==( const Vector<Type>& v1, const Vector<Type>& v2 )
{
    return ( ! ( v1 != v2 ) );
}
```

LISTING 6.1 (*Continued*)

6.1.2 Example Vector Operations

To use the **Vector** class, the user must include the VECTOR.H file (using the **#include** directive). Then the proper variables (for a scalar result) and arrays (for each vector) must be defined using a C++ **Vector** declaration. The program VECTST (shown in Listing 6.2) demonstrates the use of all six vector math functions contained in VECTOR.H using five element vectors. Both floating-point and integer arrays are used in the VECTST program. After each macro, print statements are used to display the results. No user input is required to run the VECTST program. The result of running VECTST is shown in Listing 6.3.

6.1.3 Cross Correlation and Autocorrelation

Correlation is widely used in digital signal processing because it is easy to understand and is implemented in a similar manner as an FIR filter. The cross-correlation function is maximized when the two signals are similar in frequency content and are in phase with each other. Thus, correlation is a very sensitive measure of similarity of two signals. Also, the correlation function is related to the power spectrum (or cross-power spectrum) by the Fourier transform, which makes it one of the better ways to estimate the frequency content in a signal (see Marple). Discrete correlation is simply a vector dot product defined as follows:

$$R[d] = \sum_{i=0}^{N-1} x[i]\, y[i+d] \tag{6.1}$$

Where i is the index of the arrays and d is the number of samples of shift between the x and y vectors. The correlation length is N, which determines how much data is used for each correlation result. The d variable is sometimes called the *lag value* because it indicates how many samples the y array lags the x array for the $R[d]$ correlation. When a

```
//////////////////////////////////////////////////////////////
//
// vectst.cpp - Test vector functions
//  Tests the following functions:
//   Add vectors c = a + b
//   Subtract vectors c = a - b
//   Multiply vectors c = a * b
//   Dot product c = sum( a * b )
//   Sum of vector s = sum( a )
//   Scale and copy a vector
//   Comparison ( a == b ) and ( a != b )
//
//  Inputs:
//   None
//
//  Outputs:
//   Print of test vectors and results
//
//////////////////////////////////////////////////////////////
#include "dsp.h"

//////////////////////////////////////////////////////////////
//
// int main()
//   Print test of vector functions.
//
//  Returns:
//   0 -- Success
//
//////////////////////////////////////////////////////////////
int main()
{
    try
    {
        // Initialize vector with data
        static float x[] = { 9.0f, 8.0f, 7.0f, 6.0f, 5.0f };
        static float y[] = { 1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f };
        Vector<float> v1( ELEMENTS(x), x );
        Vector<float> v2( ELEMENTS(y), y );
        Vector<float> v3;

        // Sum of two float vectors
        v3 = add( v1, v2 );
        cout << "Vector sum: v3 = v1 + v2\n";
```

LISTING 6.2 Program VECTST to the **Vector** class member functions. (*Continued*)

```
cout << "v1    v2    v3\n";
for( int i = 0 ; i < v3.length(); i++ )
    cout << v1[i] << "     " << v2[i] << "     " << v3[i] << endl;

// Difference of two float vectors
v3 = sub( v1, v2 );
cout << "\nVector subtract: v3 = v1 - v2\n";
cout << "v1    v2    v3\n";
for( i = 0 ; i < v3.length(); i++ )
    cout << v1[i] << "     " << v2[i] << "     " << v3[i] << endl;

// Product of two float vectors
v3 = pwisemult( v1, v2 );
cout << "\nVector product: v3 = v1 * v2\n";
cout << "v1    v2    v3\n";
for( i = 0 ; i < v3.length(); i++ )
    cout << v1[i] << "     " << v2[i] << "     " << v3[i] << endl;

// New vector v4 from subrange of vector v2
Vector<float> v4 = v2( 2, 5 );

// Offset product of two float vectors
v3 = pwisemult( v1, v4 );
cout << "\nVector product: v3 = v1 * v4\n";
cout << "v1    v4    v3\n";
for( i = 0 ; i < v3.length(); i++ )
    cout << v1[i] << "     " << v4[i] << "     " << v3[i] << endl;

// Dot product of two float vectors
float fDot = mult( v1, v2 );
cout << "\nDot product of v1 and v2 = " << fDot << endl;

// Sum of all elements of a float vector
float fSum = sum( v1 );
cout << "\nSum of v1 = " << fSum << endl;

// Format floating point numbers
cout.precision( 1 );
cout.setf( ios::fixed, ios::floatfield );

// Convert a vector to a different type
Vector<int> vInt( v1.length() );
conv( vInt, v1 );
cout << "\nConvert float to int: vInt = v1\n";
cout << "v1     vInt\n";
for( i = 0 ; i < vInt.length(); i++ )
```

LISTING 6.2 (*Continued*)

```
            cout << v1[i] << "     " << vInt[i] << endl;

        // Scale and convert a vector to a different type
        conv( vInt, scale( 0.5, v1 ) );
        cout << "\nScale and convert float to int: vInt = 0.5 * v1\n";
        cout << "v1     vInt\n";
        for( i = 0 ; i < vInt.length(); i++ )
            cout << v1[i] << "     " << vInt[i] << endl;

        // Set elements of a vector to a constant
        cout << "\nAll elements of a vector set to 5.0\n";
        v2 = 5.0f;
        cout << v2;

        // Compare two vectors
        v3 = v2;
        cout << v3;
        cout << ( ( v2 == v3 ) ? "Vectors equal\n" : "Vectors not equal\n" );

        // Change vector slightly
        v3[2] = -v3[2];
        cout << v3;
        cout << ( ( v2 == v3 ) ? "Vectors equal\n" : "Vectors not equal\n" );
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 6.2 (*Continued*)

number of $R[d]$ values are calculated with a series of d values, $R[d]$ is referred to as a cross-correlation function. When the x and y arrays are identical, $R[d]$ is called the *auto-correlation function*. Cross correlation is very similar to FIR filtering (see Chapter 4, Section 4.2) because the operation performed by Equation 6.1 is essentially convolution without the time reversal of the second input variable. In fact, a matched filter is simply a filter whose coefficients perform a cross correlation of the received signal with a noiseless version of the signal.

Listing 6.4 shows the COR program, which performs the cross correlation of the first record of one DSP data file with the first record of another DSP data file. The **getData** member function (which returns pointer) is used to implement the correlation indicated in Equation 6.1. The COR program prompts the user for input file names, the cor-

```
Vector sum: v3 = v1 + v2
v1    v2    v3
9     1     10
8     2     10
7     3     10
6     4     10
5     5     10

Vector subtract: v3 = v1 - v2
v1    v2    v3
9     1     8
8     2     6
7     3     4
6     4     2
5     5     0

Vector product: v3 = v1 * v2
v1    v2    v3
9     1     9
8     2     16
7     3     21
6     4     24
5     5     25

Vector product: v3 = v1 * v4
v1    v4    v3
9     3     27
8     4     32
7     5     35
6     6     36
5     7     35

Dot product of v1 and v2 = 95

Sum of v1 = 35

Convert float to int: vInt = v1
v1     vInt
9.0     9
8.0     8
7.0     7
6.0     6
5.0     5

Scale and convert float to int: vInt = 0.5 * v1
```

LISTING 6.3 The output from the VECTST program. (*Continued*)

```
v1     vInt
9.0     4
8.0     4
7.0     3
6.0     3
5.0     2

All elements of a vector set to 5.0
Vector<float>( 8 )
{
    5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0
};
Vector<float>( 8 )
{
    5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0
};
Vectors equal
Vector<float>( 8 )
{
    5.0, 5.0, -5.0, 5.0, 5.0, 5.0, 5.0, 5.0
};
Vectors not equal
```

LISTING 6.3 (*Continued*)

relation length (N), and the offset for the start of the correlation of each input signal. The **d** value is set to a series of integer values starting at a minimum lag value and continuing to a maximum lag value specified by the user. The output of the program is a single record in a DSP data file containing the correlation results. Autocorrelation can be performed by simply specifying the same input file name for the second DSP data file. For example, the following computer dialogue illustrates the autocorrelation of the Gaussian white noise sequence generated using the ADDNOISE program [see Chapter 4, Section 4.7.1 and Figure 4.26(a)]:

```
Enter file name for X signal : WNG.DAT

Enter file name for Y signal : WNG.DAT

Enter X signal offset [0...149] : 0

Enter Y signal offset [0...149] : 0

Enter correlation length [1...150] : 100

Enter minimum lag to calculate [0...50] : 0
```

```
//////////////////////////////////////////////////////////////
//
// cor.cpp - Correlation functions
//   Performs cross or auto correlation
//
//  Inputs:
//   Two DSPFiles to correlate, offsets,
//   correlation length, min and max lag values.
//
//  Outputs:
//   Correlation sums for each lage in one record
//   of a DSPFile.
//
//////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"

//////////////////////////////////////////////////////////////
//
// int main()
//
//  Returns:
//   0 -- Success
//
//////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;

        // Open X signal file
        String strX;
        do getInput( "Enter X signal file name", strX );
        while( strX.isEmpty() || dspfile.isFound( strX ) == false );

        // Read data
        Vector<float> x;
        dspfile.openRead( strX );
        dspfile.read( x );
        dspfile.close();

        // Open Y signal file
```

LISTING 6.4 Program COR used to perform cross correlation or autocorrelation of DSP data file records. (*Continued*)

```
        String strY;
        do getInput( "Enter Y signal file name", strY );
        while( strY.isEmpty() || dspfile.isFound( strY ) == false );

        // Read data
        Vector<float> y;
        dspfile.openRead( strY );
        dspfile.read( y );
        dspfile.close();

        // X and Y signal offsets for start of correlation
        int xoff = 0;
        int yoff = 0;
        getInput( "X signal offset", xoff, 0, x.length() - 1 );
        getInput( "Y signal offset", yoff, 0, y.length() - 1 );

        // Get correlation length (within limits)
        int corLength = 0;
        int minLen = min( x.length() - xoff, y.length() - yoff );
        getInput( "Correlation length", corLength, 1, minLen );

        // Lag range to use
        int lagLimit = y.length() - yoff - corLength;
        int minLag = 0;
        int maxLag = 0;
        getInput( "Minimum lag", minLag, -yoff, lagLimit );
        getInput( "Maximum lag", maxLag, minLag, lagLimit );

        // Allocate result array
        int length = maxLag - minLag + 1;
        Vector<float> cor( length );

        // Perform correlation
        for( int d = minLag, i = 0; d <= maxLag; i++, d++ )
        {
            float fcor = 0.0f;
            // Use fast read-only pointer to data
            const float *px = x.getData( corLength, xoff );
            const float *py = y.getData( corLength, yoff + d );
            for( int j = 0; j < corLength; j++ )
                fcor += *px++ * *py++;
            cor[i] = fcor;
        }

        // Write data
```

LISTING 6.4 (*Continued*)

```
        String strOut;
        do getInput( "Output file name", strOut );
        while( strOut.isEmpty() );

        // Open output file
        dspfile.openWrite( strOut );

        // write out correlation results
        dspfile.write( cor );

        // Make trailer and write to output file
        char trailer[300];
        sprintf(
            trailer,
            "File %s correlated with file %s\n"
            "X offset = %d Y offset = %d\n"
            "Correlation length = %d Minimum lag = %d Maximum lag = %d\n",
            strX, strY,
            xoff, yoff,
            corLength, minLag, maxLag);
        dspfile.setTrailer( trailer );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 6.4 (*Continued*)

```
Enter maximum lag to calculate [0...50] : 50

Enter output file name : WNGCOR.DAT
```

This autocorrelation result (file WNGCOR.DAT) is shown in Figure 6.1. Note that the autocorrelation of the sequence at a lag of zero is simply the variance of the noise times the correlation length (in this case the variance of the first 100 samples was 0.969 so that the autocorrelation at 0 is 96.9). The autocorrelation of white noise would be zero everywhere except at a lag of zero. In this example using pseudo-random noise with a 100

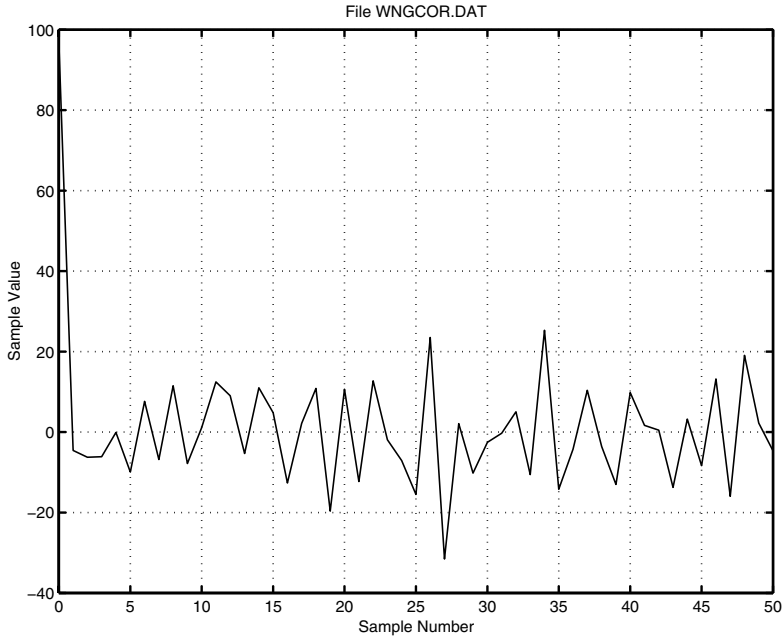


FIGURE 6.1 Autocorrelation result obtained using the COR program on the 150 Gaussian white noise samples generated using the ADDNOISE program [file WNG.DAT shown in Figure 4.26(a)].

sample correlation length, it is not zero but shows some statistical variation, which is dependent on the random number generator.

Another autocorrelation example is shown in Figure 6.2. In this case, the input signal is the radar pulse first discussed in Section 4.6.1 of Chapter 4 and shown in Figure 4.24. Note that because of the periodic nature of the radar pulse, the correlation function varies from a positive to a negative correlation with a period of three samples. Because the sampling rate is only three times the center frequency of the pulse, the correlation function shown in Figure 6.2 is not symmetrical and appears flat on the negative correlation values. A more accurate representation of the autocorrelation function of the pulse can be obtained by first 3:1 interpolating the pulse signal using the INTERP program (see Chapter 4, Section 4.5.1) and then performing an autocorrelation with a correlation length three times greater. The following computer dialogue illustrates this procedure:

```
>INTERP
Enter input file name: PULSE.DAT
```

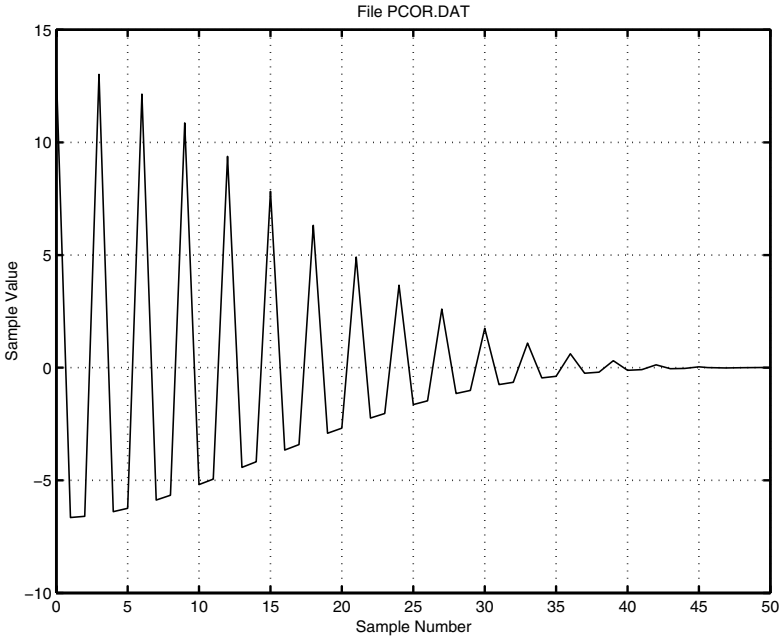


FIGURE 6.2 Autocorrelation result obtained using the COR program on the radar pulse data (file PULSE.DAT shown in Figure 4.24).

```
File trailer:
RF pulse data sampled at 3 times center frequency.

Enter interpolation ratio [2...3]: 3

Enter output file name : PULSE3.DAT

>COR

Enter file name for X signal: PULSE3.DAT
Enter file name for Y signal: PULSE3.DAT
Enter X signal offset [0...1499] : 750
Enter Y signal offset [0...1499] : 750
Enter correlation length [1...750] : 600
Enter minimum lag to calculate [-750...150] : 0
Enter maximum lag to calculate [0...150] : 150

Enter output file name : P3COR.DAT
```

Figure 6.3 shows the result of the autocorrelation of the 3:1 interpolated RADAR pulse data (file P3COR.DAT). Note that the 150-point correlation function is symmetrical and much smoother than Figure 6.2.

Cross correlation is often used to find the portion of one signal that is most similar to another signal. In some cases, the two signals are not obtained from the same sensor or they have passed through different processes. Cross correlation is illustrated in Figure 6.4 using the chirp signal generated by the **REALTIME** filtering program (discussed in Section 4.4.2 of Chapter 4). This 500-sample chirp (file RT0.DAT) is shown in Figure 4.15 of Chapter 4 and is a linear chirp from 0 to $0.5f_s$. A single 25-sample cosine wave at a frequency of $0.2f_s$ (generated using the MKWAVE program as described in Section 3.3 of Chapter 3) is cross correlated with the chirp signal. The result is shown in Figure 6.4, which shows a peak correlation near sample 200, which is where the frequency content of the chirp is $0.2f_s$. The following computer dialogue illustrates the use of the MKWAVE and COR programs used to generate Figure 6.4(b):

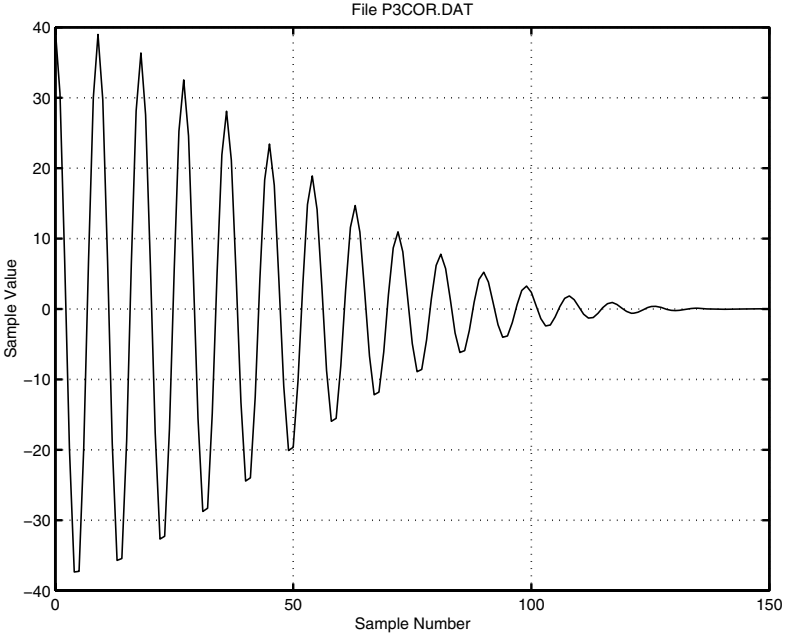


FIGURE 6.3 Autocorrelation result obtained using the INTERP and COR programs on the RADAR pulse data. The pulse data was first interpolated by a factor of 3 and then correlated.

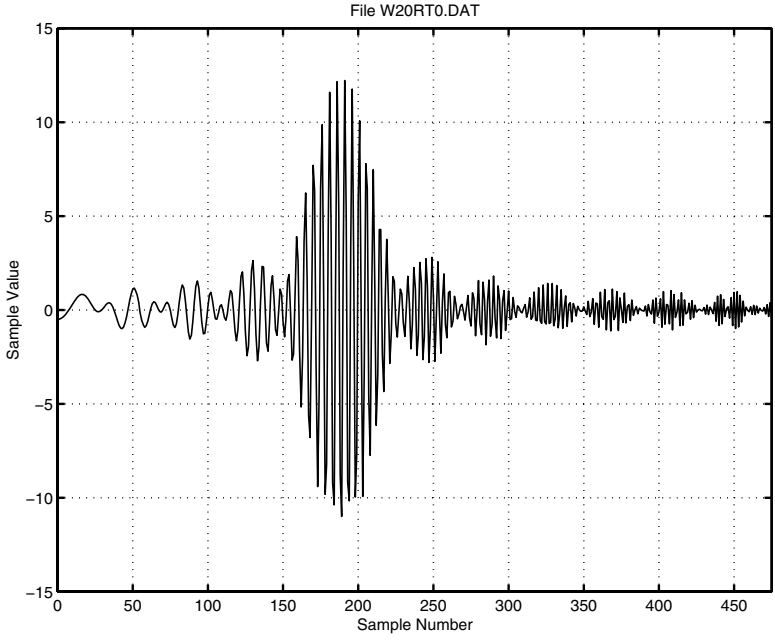


FIGURE 6.4 Cross-correlation result obtained using the COR program with the chirp signal (file RT0.DAT) and a 25-sample cosine wave at a $0.2f_s$ frequency (generated using the MKWAVE program).

```
>MKWAVE

Enter number of samples to generate [2...10000] : 25

Enter number of frequencies in sum [1...20] : 1

Enter frequency #0 [0...0.5]: 0.2

Enter file name : W20.DAT

Signal of length 25 equal to the sum of 1 cosine waves at the fol-
lowing frequencies:
f/fs = 0.200000

>COR

Enter file name for X signal : W20.DAT

Enter file name for Y signal : RT0.DAT

Enter X signal offset [0...24] : 0
```

```
Enter Y signal offset [0...499] : 0

Enter correlation length [1...25]: 25

Enter minimum lag to calculate [0...475]: 0

Enter maximum lag to calculate [0...475]: 475

Enter output file name : W20RT0.DAT
```

6.2 MATRIX OPERATIONS

A matrix is a set of numbers arranged in a two-dimensional array with some number of rows and some number of columns. As with vectors, the C++ language does not support matrix arithmetic but does support the basic data structure needed to form a matrix, namely, a two-dimensional array. This section describes the C++ functions required to perform basic manipulation of matrices. All of the matrix functions described in this section are contained in the file MATRIX.H.

The elements of a matrix are usually referenced by a row address followed by a column address. For example, the integer matrix defined by the C++ statement int **a[10][5];** could be referenced using **a[row][col],** where **row** is an integer row index from 0 to 9 and **col** is an integer column index from 0 to 4. A matrix consists of three pieces of information:

1. Type of elements (e.g., **float**, **double** or **int**)
2. Matrix size (i.e., number of rows and columns)
3. Element values

For maximum flexibility and ease of use, the following **Matrix** class will be used to hold matrix information by all matrix routines in this text:

```
////////////////////////////////////////////////////////////////
//
// Class Matix
//  Template class for matrices
//
////////////////////////////////////////////////////////////////
template <class Type>
class Matrix
{
private:
    // Data contained in rows
    Type **m_data;

    // Height and width of the matrix
```

```
    int m_rows;
    int m_cols;

public:
    ////////////////////////////////////////////////////////////
    //
    // Constructors and Destructor
    //
    ////////////////////////////////////////////////////////////
    Matrix( int rows = 0, int cols = 0 ) :
        m_data( NULL ),
        m_rows( 0 ),
        m_cols( 0 )
    {
        init( rows, cols );
    }

    // Copy constructor
    Matrix( const Matrix<Type>& m ) :
        m_data( NULL ),
        m_rows( 0 ),
        m_cols( 0 )
    {
        // Construction from existing matrix
        init( m.m_rows, m.m_cols, m.m_data );
    }

    // Destructor
    ~Matrix() { empty(); }
}
```

The two integers **m_rows** and **m_cols** in the **Matrix** structure give the dimensions of the matrix. The variable **m_data** is a pointer to an array of pointers of the data type specified. Each of the matrix functions (described in Sections 6.2.1 to 6.2.4) accepts one or more references to **Matrix** structures and returns a pointer to a result **Matrix** class (except for **det**, which returns a **double** value). Also shown in the above code segment is the **Matrix** class constructor, which allocates and initializes a matrix by calling a **private** member function **init** (see MATRIX.H). For example, the following code segment allocates a 10 by 5 **float** matrix called **A:**

```
Matrix<float> A( 10 , 5);
```

6.2.1 Matrix Element Manipulation

The elements of a matrix must be displayed, scaled, and moved from one place to another to implement the solution of matrix problems. The member functions defined in MATRIX.H are designed to perform some of the most common matrix element manipu-

lations. The code shown in Listing 6.5 overloads the =, [ ], ( ) to set and get matrix data. The operator **(row, col, rows, cols)** can be used to extract a submatrix (size **rows** by **cols**) from a larger matrix. Example use of these operators will be shown in Section 6.2.5.

```
//////////////////////////////////////////////////////////////
//
// Get and Set matrix contents
//
//////////////////////////////////////////////////////////////

    // Set matrix equal to matrix m
    Matrix<Type>& operator=( const Matrix<Type>& m )
    {
        // Equal to itself
        if( &m != this )
                init( m.m_rows, m.m_cols, m.m_data );

        return *this;
    }

    // Set all elements of matrix equal to constant
    Matrix<Type>& operator=( const Type t )
    {
        for( int i = 0; i < m_rows; i++ )
        {
            Type *pt = m_data[i];
            for( int j = 0; j < m_cols; j++ )
                    *pt++ = t;
        }
        return *this;
    }

    // Return a matrix row
    Type* operator[]( int row )
    {
        if( isEmpty() )
                throw DSPBoundaryException( "Matrix has no data" );

        if( row < 0 || row >= m_rows )
            throw DSPBoundaryException();

        return m_data[row];
    }

    const Type* operator[]( int row ) const
```

LISTING 6.5 Functions to manipulate data associated with a matrix class. (*Continued*)

```
    {
        if( isEmpty() )
                throw DSPBoundaryException( "Matrix has no data" );

        if( row < 0 || row >= m_rows )
                throw DSPBoundaryException();
        return m_data[row];
    }

    // Swap two rows
    void swapRows( int r1, int r2 )
    {
        if( isEmpty() )
                throw DSPBoundaryException( "Matrix has no data" );

        if( r1 < 0 || r1 >= m_rows || r2 < 0 || r2 >= m_rows )
                throw DSPBoundaryException();

        Type *pt = m_data[r1];
        m_data[r1] = m_data[r2];
        m_data[r2] = pt;
    }

    // Return a new read-only submatrix of this matrix
    const Matrix<Type> operator()( int row, int col, int rows, int cols ) const
    {
        // Validate parameters
        if( isEmpty() )
                throw DSPBoundaryException( "Matrix has no data" );

        if( row < 0 || rows < 1 || row + rows > m_rows )
                throw DSPBoundaryException( "Row out of range" );

        if( col < 0 || cols < 1 || col + cols > m_cols )
                throw DSPBoundaryException( "Column out of range" );

        Matrix<Type> m( rows, cols );
        for( int i = 0; i < rows; i++ )
        {
                for( int j = 0; j < cols; j++ )
                        m[i][j] = m_data[i + row][j + col];
        }

        return m;
    }
     // DSPFile reads and writes m_data for fast disk access
    friend class DSPFile;
};
```

LISTING 6.5 (*Continued*)

The matrix print function is implemented in C++ by overloading the **<<** as follows:

```
////////////////////////////////////////////////////////////////////
//
// ostream& operator<<( ostream& os, const Matrix<Type>& m )
//   Writes the matrix out to an ostream for display.
//
//  Returns:
//   ostream after write
//
////////////////////////////////////////////////////////////////////
template <class Type>
ostream& operator<<( ostream& os, const Matrix<Type>& m )
{
    int colsPerLine = 5;
    // Check if small type
    if( sizeof( Type ) <= 2 )
        colsPerLine = 10;
    // Check if double type
    else if( typeid( Type ) == typeid( double ) )
        colsPerLine = 4;

    // Format for display and print
    char szFmt[64];
    for( int i = 0; i < m.rows(); i++ )
    {
        for( int j = 0; j < m.cols(); j++ )
        {
            if( ( j % colsPerLine ) == 0 )
            {
                if( j == 0 )
                    sprintf( szFmt, "Row%3d:", i );
                else
                    sprintf( szFmt, "\n [%3d]", j );
                os << szFmt;
            }

            // Format depends on type
            if( typeid( Type ) == typeid( float ) )
                sprintf( szFmt, "%14.5g", m[i][j] );
            else if( typeid( Type ) == typeid( double ) )
                sprintf( szFmt, "%18.11lg", m[i][j] );
            else
                sprintf( szFmt, "%7d", m[i][j] );
            os << szFmt;
        }
        os << endl;
    }
    return os;
}
```

This code segment displays the elements of a matrix one row at a time with the row number displayed at the beginning of each line. If a large number of columns are present in the matrix, the code places the data on multiple lines with the column number at the beginning of each line.

The function **transpose** the data elements are rearranged such that the row and column indices are interchanged. The following C++ code is used to do this for matrices:

```
////////////////////////////////////////////////////////////////
//
// Matrix<Type> transpose( const Matrix<Type>& m )
//   Transpose elements of a matrix.
//
//  Throws:
//   DSPException
//
//  Returns:
//   Resultant matrix ( mT )
//
////////////////////////////////////////////////////////////////
template <class Type>
Matrix<Type> transpose( const Matrix<Type>& m )
{
    int rows = m.rows();
    int cols = m.cols();
    if( rows == 0 || cols == 0 )
      throw DSPBoundaryException( "Matrix has no data" );

    Matrix<Type> mRet( cols, rows );

    for( int i = 0; i < rows; i++ )
      for( int j = 0; j < cols; j++ )
             mRet[j][i] = m[i][j];
    return mRet;
}
```

6.2.2 Matrix Arithmetic

Arithmetic operations on matrices are very similar to the arithmetic operations on vectors. When two matrices are added or subtracted, the operation is performed on an element-by-element basis. The result of the addition or subtraction of each pair of elements of the two operands forms the result matrix. The functions **add** and **sub** perform matrix addition or subtraction of a pair of **Matrix** structures and return a result **Matrix** structure. Matrix multiplication can be element-by-element multiplication or a cross product.

Element-by-element multiplication is performed by the function **pwisemult** (for matrix multiply point-wise) and the cross product is calculated using the **mult** function. Listing 6.6 shows the complete **add**, **sub**, and **pwisemult** functions, which use C++ member functions to implement all element type combinations with one set of functions (see the MATRIX.H file).

```
////////////////////////////////////////////////////////////////
//
// Matrix<Type> add( const Matrix<Type>& m1, const Matrix<Type>& m2 )
//   Add two matrices.
//
//  Throws:
//   DSPException
//
//  Returns:
//   Resultant matrix ( m1 + m2 )
//
////////////////////////////////////////////////////////////////
template <class Type>
Matrix<Type> add( const Matrix<Type>& m1, const Matrix<Type>& m2 )
{
    int rows = min( m1.rows(), m2.rows() );
    int cols = min( m1.cols(), m2.cols() );
    if( rows == 0 || cols == 0 )
        throw DSPBoundaryException( "Matrix has no data" );
    Matrix<Type> mRet( rows, cols );
    for( int i = 0; i < rows; i++ )
        for( int j = 0; j < cols; j++ )
            mRet[i][j] = m1[i][j] + m2[i][j];
    return mRet;
}

////////////////////////////////////////////////////////////////
//
// Matrix<Type> sub( const Matrix<Type>& m1, const Matrix<Type>& 2 )
//   Subtract two matrices.
//
//  Throws:
//   DSPException
//
//  Returns:
```

LISTING 6.6 Member functions used to add, subtract, and multiply two matrices element by element. (*Continued*)

```
//   Resultant matrix ( m1 - m2 )
//
////////////////////////////////////////////////////////////////
template <class Type>
Matrix<Type> sub( const Matrix<Type>& m1, const Matrix<Type>& m2 )
{
    int rows = min( m1.rows(), m2.rows() );
    int cols = min( m1.cols(), m2.cols() );
    if( rows == 0 || cols == 0 )
        throw DSPBoundaryException( "Matrix has no data" );
    Matrix<Type> mRet( rows, cols );
    for( int i = 0; i < rows; i++ )
        for( int j = 0; j < cols; j++ )
            mRet[i][j] = m1[i][j] - m2[i][j];
    return mRet;
}
////////////////////////////////////////////////////////////////
//
// Matrix<Type> pwisemult( const Matrix<Type>& m1, const Matrix<Type>& m2 )
//   Multiply elements of two matrices.
//
//  Throws:
//   DSPException
//
//  Returns:
//   Resultant matrix ( m1 * m2 )
//
////////////////////////////////////////////////////////////////
template <class Type>
Matrix<Type> pwisemult( const Matrix<Type>& m1, const Matrix<Type>& m2 )
{
    int rows = min( m1.rows(), m2.rows() );
    int cols = min( m1.cols(), m2.cols() );
    if( rows == 0 || cols == 0 )
        throw DSPBoundaryException( "Matrix has no data" );
    Matrix<Type> mRet( rows, cols );
    for( int i = 0; i < rows; i++ )
        for( int j = 0; j < cols; j++ )
            mRet[i][j] = m1[i][j] * m2[i][j];
    return mRet;
}
```

LISTING 6.6 (*Continued*)

The matrix cross product calculated by **mult** is calculated using pointers as was described in Section 2.7.3 of Chapter 2. The cross product is defined only if the number of columns of the first matrix are the same as the number of rows in the second matrix. If this is not the case, the **mult** function throws an exception. The following cross-product macro is used in the **mult** function (see the MATRIX.H file):

```
////////////////////////////////////////////////////////////////
//
// Matrix<Type> mult( const Matrix<Type>& m1, const Matrix<Type>& m2 )
//   Cross product of two matrices.
//
//  Throws:
//   DSPException
//
//  Returns:
//   Resultant matrix ( m1 cross m2 )
//
////////////////////////////////////////////////////////////////
template <class Type>
Matrix<Type> mult( const Matrix<Type>& m1, const Matrix<Type>& m2 )
{
    int rows = m1.rows();
    int cols = m2.cols();
    if( rows == 0 || cols == 0 )
        throw DSPBoundaryException( "Matrix has no data" );
    if( m1.cols() != m2.rows() )
       throw DSPBoundaryException( "Row size does not match column size" );

    Matrix<Type> mRet( rows, cols );
    for( int i = 0; i < rows; i++ )
    {
       for( int j = 0; j < cols; j++ )
       {
              Type tsum = 0;
              for( int k = 0; k < m1.cols(); k++ )
                      tsum += m1[i][k] * m2[k][j];
              mRet[i][j] = tsum;
        }
    }
    return mRet;
}
```

6.2.3 Matrix Inversion

Matrix inversion can be used to solve a set of linear equations, **Ax = b,** by finding the inverse of **A** and multiplying the resulting inverse by the constant vector **b. A** is normally a nonsingular square matrix giving the coefficients of the linear equations, and **x** is the vec-

tor of the unknowns. If the solution of another set of equations with the same **A** and a different **b** vector is desired, the new solution can be found using the same matrix inverse by performing a matrix multiply with the new **b** vector. For large **A** matrices, the matrix inverse is often difficult to calculate accurately. The **invert** function shown in Listing 6.7 uses double-precision arithmetic and the Gauss-Jordan elimination algorithm with full

```
//////////////////////////////////////////////////////////////
//
// Matrix<Type> invert( const Matrix<Type>& m )
//   Invert a square matrix.
//
//  Throws:
//   DSPException
//
//  Returns:
//   Resultant matrix ( mI )
//
//////////////////////////////////////////////////////////////
template <class Type>
Matrix<double> invert( const Matrix<Type>& m )
{
    if( m.isEmpty() )
        throw DSPBoundaryException( "No data in matrix" );

    // Size of square matrix
    int n = m.rows();

    // Check for NxN matrix
    if( n != m.cols() )
        throw DSPBoundaryException( "invert of non-square matrix" );

    // Allocate matrix for the inverse
    Matrix<double> mI( n, n );

    // Convert to double matrix
    conv( mI, m );

    // Allocate index arrays and set to zero
    int *pivotFlag = new int[n];
    int *swapRow = new int[n];
    int *swapCol = new int[n];
    if( pivotFlag == NULL || swapRow == NULL || swapCol == NULL )
        throw DSPMemoryException();
```

LISTING 6.7 Function **invert** used to invert a matrix using Gauss-Jordan elimination with full pivoting. (*Continued*)

```
    for( int i = 0; i < n; i++ )
    {
        pivotFlag[i] = 0;
        swapRow[i] = 0;
        swapCol[i] = 0;
    }


    // Pivoting n iterations
    int row, irow, col, icol;
    for( i = 0; i < n; i++ )
    {
        // Find the biggest pivot element
        double big = 0.0;
        for( row = 0; row < n; row++ )
        {
            if( pivotFlag[row] == 0 )
            {
                // Only unused pivots
                for( col = 0; col < n; col++ )
                {
                        if( pivotFlag[col] == 0 )
                        {
                                double abs_element = fabs( mI[row][col] );
                                if( abs_element >= big )
                                {
                                       big = abs_element;
                                       irow = row;
                                       icol = col;
                                }
                        }
                }
            }
        }

        // Mark this pivot as used
        pivotFlag[icol]++;

        // Swap rows to make this diagonal the biggest absolute pivot
        if( irow != icol )
        {
            for( col = 0; col < n; col++ )
            {
                double temp = mI[irow][col];
                mI[irow][col] = mI[icol][col];
                mI[icol][col] = temp;
```

LISTING 6.7 (*Continued*)

```
            }
        }

        // Store what we swaped
        swapRow[i] = irow;
        swapCol[i] = icol;

        // Bad news if the pivot is zero
        if( mI[icol][icol] == 0.0 )
            throw DSPMathException( "Invert of singular matrix" );

        // Divide the row by the pivot
        double pivotInverse = 1.0 / mI[icol][icol];

        // Pivot = 1 to avoid round off
        mI[icol][icol] = 1.0;
        for( col = 0; col < n; col++ )
            mI[icol][col] *= pivotInverse;

        // Fix the other rows by subtracting
        for( row = 0; row < n; row++ )
        {
            if( row != icol )
            {
                double temp = mI[row][icol];
                mI[row][icol] = 0.0;
                for( col = 0; col < n; col++)
                        mI[row][col] -= mI[icol][col] * temp;
            }
        }
    }

    // Fix the effect of all the swaps for final answer
    for( int swap = n-1; swap >= 0; swap— )
    {
        if( swapRow[swap] != swapCol[swap] )
        {
            for( row = 0; row < n; row++ )
            {
                double temp = mI[row][swapRow[swap]];
                mI[row][swapRow[swap]] = mI[row][swapCol[swap]];
                mI[row][swapCol[swap]] = temp;
            }
        }
    }
```

LISTING 6.7 (*Continued*)

```
    // Free memory
    delete [] pivotFlag;
    delete [] swapRow;
    delete [] swapCol;

    // Convert back to Type
    return mI;
}
```

LISTING 6.7 (*Continued*)

pivoting, which results in a numerically stable and highly accurate matrix inverse. The Gauss-Jordan elimination algorithm is straightforward, efficient, and as numerically stable as other methods. It is related to the more familiar Gaussian elimination algorithm, which can be used to solve a set of linear equations but usually does not reduce the **A** matrix all the way to form the matrix inverse. If the matrix inverse is not required, the solution of a single set of linear equations can be calculated more rapidly and with greater accuracy than Gauss-Jordan or Gaussian elimination using alternate methods (see References).

The Gauss-Jordan elimination algorithm is numerically unstable in the presence of any round-off error unless some type of pivoting is used. *Pivoting* is interchanging the rows and columns (in full pivoting) of the matrix such that the largest magnitude element in the matrix is used as the diagonal element. This diagonal element (the pivot) is used to divide the other elements in the row, which are then used to eliminate the other rows. The change in the answer due to interchanging the rows and columns must be corrected after the elimination process is complete. The algorithm implemented by **invert** consists of the following steps:

1. Create a square matrix equal to the input matrix with double-precision elements. The Gauss-Jordan algorithm is implemented as an in-place algorithm so that the resulting inverse will appear in this double-precision matrix (variable **mI**).
2. Find the largest magnitude element in the matrix that has not been used as a pivot previously. Interchange the rows and columns required to make the largest element in the matrix the current diagonal pivot element. Store what was swapped in the **swapRow** and **swapCol** arrays.
3. Divide the row by the pivot value found in step 2.
4. Perform the elimination by subtracting a portion of the pivot from the rest of the rows. Set the pivot element to unity.
5. Repeat steps 2 through 4 for each row of the matrix (**n** iterations). If any of pivots are equal to zero, give an error message indicating that the matrix is singular.
6. Fix the affect of the pivoting in step 2 by interchanging the appropriate columns, which were stored in the **swapRow** and **swapCol** arrays in step 2.

7. Free the temporary arrays allocated by **invert** (**pivotFlag, swapRow,** and **swapCol**) and return the inverse **(mI).**

6.2.4 Matrix Determinant

Listing 6.8 shows the function **det,** which determines the determinant of the matrix passed to it. It returns a single, double-precision scalar value. The **det** function is similar to the **invert** function because a row-reduction operation is used. In this case, however, only the elements above the diagonal must be eliminated because the determinant can be calculated from the product of the diagonal elements after the elements above or below

```
////////////////////////////////////////////////////////////////
//
// double det( const Matrix<Type>& m )
//   Calculate determinant of a matrix.
//
//  Throws:
//   DSPException
//
//  Returns:
//   Determinant as double.
//
////////////////////////////////////////////////////////////////
template <class Type>
double det( const Matrix<Type>& m )
{
    if( m.isEmpty() )
        throw DSPBoundaryException( "No data in matrix" );

    // Size of square matrix
    int n = m.rows();

    // Check for NxN matrix
    if( n != m.cols() )
        throw DSPBoundaryException( "Determinant of non-square matrix" );

    // Allocate space for the determinant calculation matrix
    Matrix<double> mDet( n, n );

    // Copy to double matrix for calculations
    conv( mDet, m );
```

LISTING 6.8 Function **det** used to find the determinant of a matrix using upper factorization to find the diagonal values to form the result. (*Continued*)

```
    // Initialize the answer
    double det = 1.0;
    for( int pivot = 0; pivot < n-1; pivot++ )
    {
        // Find the biggest absolute pivot
        double big = fabs( mDet[pivot][pivot] );

        // Initialize for no swap
        int swapRow = 0;
        for( int row = pivot + 1; row < n; row++ )
        {
            double abs_element = fabs( mDet[row][pivot] );
            if( abs_element > big )
            {
                swapRow = row;
                big = abs_element;
            }
        }

        // Unless swapRow is still zero we must swap two rows
        if( swapRow != 0 )
        {
            // Swap two rows
            mDet.swapRows( pivot, swapRow );

            // Change the sign of determinant because of swap
            det *= -mDet[pivot][pivot];
        }
        else
        {
            // Calculate the determinant by the product of the pivots
            det *= mDet[pivot][pivot];
        }
        // If almost singular matrix, give up now
        if( fabs( det ) < 1.0e-50 )
            return det;

        double pivotInverse = 1.0 / mDet[pivot][pivot];
        for( int col = pivot + 1; col < n; col++ )
            mDet[pivot][col] = mDet[pivot][col] * pivotInverse;

        for( row = pivot + 1; row < n; row++ )
        {
            double temp = mDet[row][pivot];
            for( col = pivot + 1; col < n; col++ )
```

LISTING 6.8 (*Continued*)

```
            {
                mDet[row][col] = mDet[row][col] - mDet[pivot][col] * temp;
            }
        }
    }

    // Last pivot, no reduction required
    det *= mDet[n-1][n-1];
    return det;
}
```

LISTING 6.8 (*Continued*)

the diagonal are row reduced to zero. The method used to calculate a determinant in this way is called LU decomposition for lower-upper decomposition. In the complete LU decomposition algorithm, the input matrix is factored into two matrices, one with zero elements below the diagonal and one with zero elements above the diagonal (see References).

The **det** function determines only the lower factor in the LU decomposition where the elements above the diagonal are zero. In fact, all of the lower factors are not calculated, since only the diagonal elements are used to form the determinant. Full pivoting is again implemented and because only the sign of the determinant changes when rows or columns are swapped, no storage of the pivoting history is required.

6.2.5 Example Matrix Routine

The program MATTST shown in Listing 6.9 demonstrates the matrix routines. Listing 6.10 shows the output from the MATTST program. The solutions to three different matrix problems are illustrated in the MATTST program. Both integer and floating-point matrices are used in order to test the matrix functions as completely as possible. Further use of the matrix routines is demonstrated by the least squares curve-fitting program discussed in Section 6.4.1.

In the first example in MATTST is the solution of a simple 3×3 matrix equation ($\mathbf{Ax} = \mathbf{b}$). It illustrates the basic use of the **invert** and **mult** functions. In this case, the matrix **A** and the column vector **b** are both integer matrices. In the second example, the same 3×3 matrix equation is solved using Crammer's rule in order to illustrate the **det** function. The solutions obtained by matrix inversion and Crammer's are nearly identical. Finally, a series of Hilbert matrices (an example of a poorly conditioned matrix) are inverted and compared to the known Hilbert matrix inverse. The last example tests the accuracy of the **invert** function and verifies the accuracy of the double-precision arithmetic implemented by the target computer and compiler. The details of the three matrix examples in the MATTST program are as follows:

```
////////////////////////////////////////////////////////////////
//
// mattst.cpp - Test matrix functions
//   Tests the following functions:
//    Add matrices c = a + b
//    Subtract matrices c = a - b
//    Multiply matrices c = a * b
//    Sum of matrix x = sum( a )
//    Scale matrix c = scale * a
//    Transpose matrix c = cT
//    Invert matrix c = c^-1
//    Submatrix from matrix c = a( row, col, rows, cols )
//    Coversion of matrix to another type
//    Matrix determinant x = det ( c )
//    Comparison ( a == b ) and ( a != b )
//
//  Inputs:
//   None
//
//  Outputs:
//   Print of test matrices and results
//
////////////////////////////////////////////////////////////////
#include "dsp.h"

////////////////////////////////////////////////////////////////
//
// int main()
//   Matrix test functions.
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        // To convert to double
        Matrix<double> Aconv( 3, 3 );

        // 3x3 short int matrix
        Matrix<short> A( 3, 3 );
        A[0][0] = -1;    A[0][1] = 2;    A[0][2] = 1;
        A[1][0] = -2;    A[1][1] = 2;    A[1][2] = 1;
```

LISTING 6.9 Program MATTST used to demonstrate and test the matrix routines. (*Continued*)

```
A[2][0] = -1;    A[2][1] = 2;    A[2][2] = 3;
cout << "A matrix:\n" << A << endl;

// Sub matrix
cout << "Submatrix A( 1, 1, 2, 2 ):\n" << A( 1, 1, 2, 2 );
// Invert matrix (always returns double)
Matrix<double> Ai = invert( A );
cout << "\nA matrix inverse (Ai):\n" << Ai << endl;

// Get cross product
cout << "Product of A and A inverse (should be I):\n";
cout << mult( conv( Aconv, A ), Ai ) << endl;

// Matrix as a vector
Matrix<double> B( 3, 1 );
B[0][0] = 5.0;
B[1][0] = 6.0;
B[2][0] = 9.0;

cout << "B vector:\n" << B << endl;

cout << "Result ( x = Ai * b ):\n" << mult( Ai, B ) << endl;

// 3x3 short int matrix
Matrix<short> C( 3, 3 );
C[0][0] = 1;    C[0][1] = 4;    C[0][2] = 0;
C[1][0] = 2;    C[1][1] = 7;    C[1][2] = 1;
C[2][0] = 3;    C[2][1] = 3;    C[2][2] = 3;
cout << “C matrix:\n” << C << endl;

// Arithmetic
cout << "C + A:\n" << add( C, A ) << endl;
cout << "C - A:\n" << sub( C, A ) << endl;
cout << "C * A:\n" << pwisemult( C, A ) << endl;

cout << "Sum of C: " << sum( C ) << endl << endl;
cout << "Scale C by 2.5:\n" << scale( 2.5f, C ) << endl;

// Compare two matrices
C = A;
cout
    << A << C << endl
    << ( ( A == C ) ? "Matrices equal\n" : "Matrices not equal\n" )
    << endl;

// Change vector slightly
C[1][1] = -C[1][1];
```

LISTING 6.9 (*Continued*)

```
        cout
            << C << endl
            << ( ( A == C ) ? "Matrices equal\n" : "Matrices not equal\n" )
            << endl;

        // Crammer's rule test of determinant and transpose
        double denom = det( A );
        cout << "Crammer's rule denominator, det(A) = " << denom << endl;

        Matrix<short> Bint;
        conv( Bint, B );
        Vector<short> b = Bint.getCol( 0 );
        for(int i = 0 ; i < 3 ; i++) {
            Matrix<short> At = transpose( A );
            At.setRow( i, b );
            cout << "Crammer's rule x[" << i << "] = ";
            cout << det( At ) / denom << endl;
        }

        At.swapRows( 0, 1 );
        cout << "Swap rows 0 and 1 of At:\n" << At << endl;
        cout << "Determinant of At = " << det( At ) << endl << endl << endl;

        // Invert and find the determinant of Hilbert matrices 2 to 6
        for( int n = 2; n < 7; n++ )
        {
            Matrix<double> H( n, n );

            for( int i = 0; i < n ; i++ )
                for( int j = 0; j < n ; j++ )
                    H[i][j] = 1.0/(double)( i+j+1 );

            cout << "Hilbert matrix N = " << i << endl << H;
            cout << "Hilbert determinant = " << det( H ) << endl;
            Matrix<double> Hi = invert( H );
            cout << "Calculated Hilbert matrix inverse N = " << i << endl;
            cout << Hi;

            // Make exact Hilbert inverse matrix
            Matrix<double> Hie( n, n );
            long diag = 0;
            long rest = 0;
            for( i = 0; i < n; i++ )
            {
                if( i == 0 )
                    diag = n;
                else
```

LISTING 6.9 (*Continued*)

```
                    diag = ((n-i)*diag*(n+i))/(i*i);

                rest = diag*diag;
                Hie[i][i] = rest/(2*i+1);
                for( int j = i + 1 ; j < n ; j++ )
                {
                    rest = -((n-j)*rest*(n+j))/(j*j);
                    Hie[i][j] = Hie[j][i] = rest/(i+j+1);
                }
            }

            // Difference of exact inverse and the calculated one
            cout << "Error in Hilbert matrix inverse N = " << i << endl;
            cout << sub( Hi, Hie ) << endl;
        }
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 6.9 (*Continued*)

```
A matrix:
Row 0:     -1     2     1
Row 1:     -2     2     1
Row 2:     -1     2     3

Submatrix A( 1, 1, 2, 2 ):
Row 0:      2     1
Row 1:      2     3

A matrix inverse (Ai):
Row 0:                 1               -1                0
Row 1:              1.25             -0.5            -0.25
Row 2:              -0.5                0              0.5

Product of A and A inverse (should be I):
Row 0:                 1                0                0
Row 1: 5.5511151231e-017                1                0
Row 2: 2.2204460493e-016                0                1
```

LISTING 6.10 The output from program MATTST (see Listing 6.9). (*Continued*)

```
B vector:
Row 0:                    5
Row 1:                    6
Row 2:                    9

Result ( x = Ai * b ):
Row 0:                   -1
Row 1:                    1
Row 2:                    2

C matrix:
Row 0:       1       4      0
Row 1:       2       7      1
Row 2:       3       3      3

C + A:
Row 0:       0       6      1
Row 1:       0       9      2
Row 2:       2       5      6

C - A:
Row 0:       2       2     -1
Row 1:       4       5      0
Row 2:       4       1      0

C * A:
Row 0:      -1       8      0
Row 1:      -4      14      1
Row 2:      -3       6      9

Sum of C: 24

Scale C by 2.5:
Row 0:       2      10      0
Row 1:       5      17      2
Row 2:       7       7      7

Row 0:      -1       2      1
Row 1:      -2       2      1
Row 2:      -1       2      3
Row 0:      -1       2      1
Row 1:      -2       2      1
Row 2:      -1       2      3

Matrices equal

Row 0:      -1       2      1
Row 1:      -2      -2      1
```

LISTING 6.10 (*Continued*)

```
Row 2:     -1     2     3

Matrices not equal

Crammer's rule denominator, det(A) = 4
Crammer's rule x[0] = -1
Crammer's rule x[1] = 1
Crammer's rule x[2] = 2

Hilbert matrix N = 2
Row 0:                 1               0.5
Row 1:               0.5     0.33333333333
Hilbert determinant = 0.0833333
Calculated Hilbert matrix inverse N = 2
Row 0:                 4                -6
Row 1:                -6                12
Error in Hilbert matrix inverse N = 2
Row 0: 8.881784197e-016 -1.7763568394e-015
Row 1:-1.7763568394e-015 3.5527136788e-015

Hilbert matrix N = 3
Row 0:                 1               0.5       0.33333333333
Row 1:               0.5     0.33333333333                0.25
Row 2:     0.33333333333              0.25                 0.2
Hilbert determinant = 0.000462963
Calculated Hilbert matrix inverse N = 3
Row 0:                 9               -36                  30
Row 1:               -36               192                -180
Row 2:                30              -180                 180
Error in Hilbert matrix inverse N = 3
Row 0: 4.4408920985e-014 -2.2737367544e-013  2.0605739337e-013
Row 1: -2.344791028e-013  1.1937117961e-012 -1.1084466678e-012
Row 2: 2.2026824809e-013 -1.1084466678e-012  1.0516032489e-012

Hilbert matrix N = 4
Row 0:                 1               0.5       0.33333333333             0.25
Row 1:               0.5           3333333                0.25              0.2
Row 2:     0.33333333333              0.25                 0.2    0.16666666667
Row 3:              0.25               0.2       0.16666666667    0.14285714286
Hilbert determinant = 1.65344e-007
Calculated Hilbert matrix inverse N = 4
Row 0:   16         -120               240                -140
Row 1: -120         1200             -2700                1680
Row 2:  240        -2700              6480               -4200
Row 3: -140         1680             -4200                2800
```

LISTING 6.10 (*Continued*)

```
Error in Hilbert matrix inverse N = 4
Row 0:  -4.86721774e-013   6.6506800067e-012 -1.7365664462e-011  1.1851852832e-011
Row 1:  6.139089237e-012  -8.1399775809e-011  2.1009327611e-010 -1.4233592083e-010
Row 2:-1.5518253349e-011    2.037268132e-010 -5.2204995882e-010  3.5197444959e-010
Row 3: 1.0373923942e-011  -1.3483258954e-010  3.4469849197e-010 -2.323758963e-010

Hilbert matrix N = 5
Row 0:                 1                0.5      0.33333333333               0.25
  [ 4]               0.2
Row 1: 0.5 0.33333333333               0.25                0.2
  [ 4]     0.16666666667
Row 2:     0.33333333333               0.25                0.2      0.16666666667
  [ 4]     0.14285714286
Row 3:              0.25                0.2      0.16666666667      0.14285714286
  [ 4]             0.125
Row 4: 0.2 0.16666666667      0.14285714286               0.125
  [ 4]     0.11111111111
Hilbert determinant = 3.7493e-012
Calculated Hilbert matrix inverse N = 5
Row 0:                25               -300               1050              -1400
  [ 4]               630
Row 1:              -300               4800             -18900              26880
  [ 4]            -12600
Row 2:              1050             -18900              79380            -117600
  [ 4]             56700
Row 3:             -1400              26880            -117600             179200
  [ 4]            -88200
Row 4:               630             -12600              56700             -88200
  [ 4]             44100
Error in Hilbert matrix inverse N = 5
Row 0:-1.6342482922e-011  2.8558133636e-010 -1.1655174603e-009  1.6859758034e-009
  [ 4]-7.9671735875e-010
Row 1: 2.8626345738e-010 -4.9194568419e-009  1.9841536414e-008 -2.8448994271e-008
  [ 4] 1.3345925254e-008
Row 2: -1.183252607e-009  2.0121660782e-008 -8.0530298874e-008  1.1475640349e-007
  [ 4]-5.3565599956e-008
Row 3: 1.7350885173e-009 -2.9282091418e-008  1.1654628906e-007 -1.6533886082e-007
  [ 4] 7.6906871982e-008
Row 4: -8.3036866272e-010 1.3935277821e-008 -5.5231794249e-008 7 .8085577115e-008
  [ 4]-3.6219717003e-008

Hilbert matrix N = 6
Row 0:                 1                0.5      0.33333333333               0.25
  [ 4]               0.2     0.16666666667
```

LISTING 6.10 (*Continued*)

```
Row 1:               0.5      0.33333333333                0.25               0.2
  [ 4]     0.16666666667      0.14285714286
Row 2:     0.33333333333               0.25                 0.2     0.16666666667
  [ 4]     0.14285714286              0.125
Row 3:              0.25                0.2        0.16666666667     0.14285714286
  [ 4]             0.125      0.11111111111
Row 4:               0.2      0.16666666667        0.14285714286             0.125
  [ 4]     0.11111111111                0.1
Row 5:     0.16666666667      0.14285714286                0.125     0.11111111111
  [ 4]               0.1     0.090909090909
Hilbert determinant = 5.3673e-018
Calculated Hilbert matrix inverse N = 6
Row 0:      36.000000001      -630.00000004         3360.0000003     -7560.0000008
  [ 4]      7560.0000009      -2772.0000004
Row 1:     -630.00000004       14700.000001        -88200.000009      211680.00002
  [ 4]     -220500.00003       83160.000011
Row 2:      3360.0000003      -88200.000009         564480.00006     -1411200.0002
  [ 4]      1512000.0002      -582120.00007
Row 3:     -7560.0000008       211680.00002        -1411200.0002      3628800.0004
  [ 4]     -3969000.0005       1552320.0002
Row 4:      7560.0000009      -220500.00003         1512000.0002     -3969000.0005
  [ 4]      4410000.0005      -1746360.0002
Row 5:     -2772.0000003        83160.00001        -582120.00007      1552320.0002
  [ 4]     -1746360.0002       698544.00009
Error in Hilbert matrix inverse N = 6
Row 0: 1.4691465822e-009 -4.4311718739e-008   3.101981747e-007-8.  2591850514e-007
  [ 4] 9.2731443146e-007 -3.7017389332e-007
Row 1:-4.3339014155e-008   1.304792022e-006 -9.1234833235e-006 2.  4272536393e-005
  [ 4]-2.7236732421e-005  1.0867865058e-005
Row 2: 2.9902457754e-007 -8.9926761575e-006    6.2836101279e-005 -0.00016709370539
  [ 4]  0.00018743542023 -7.4770185165e-005
Row 3:-7.8815264715e-007  2.3684697226e-005    -0.00016541942023  0.00043974351138
  [ 4] -0.00049316463992   0.00019669509493
Row 4: 8.7833086582e-007 -2.6380294003e-005     0.00018418394029 -0.00048951525241
  [ 4]  0.00054889265448  -0.00021889456548
Row 5:-3.4861523091e-007   1.046617399e-005   -7.3054805398e-005  0.00019412813708
  [ 4] -0.00021764845587  8.6788553745e-005
```

LISTING 6.10 (*Continued*)

1. Matrix equation solution example—the following 3×3 matrix equation ($\mathbf{Ax} = \mathbf{b}$) is solved for $\mathbf{x}$ by finding the matrix inverse:

$$\begin{bmatrix} -1 & 2 & 1 \\ -2 & 2 & 1 \\ -1 & 2 & 3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \\ 9 \end{bmatrix} \tag{6.2}$$

The exact inverse of $\mathbf{A}$ is

$$\mathbf{A}^{-1} = \begin{bmatrix} 1 & -1 & 0 \\ 5/4 & -1/2 & -1/4 \\ -1/2 & 0 & 1/2 \end{bmatrix}$$

and the exact solution is

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix}$$

The product $\mathbf{AA}^{-1}$ is also formed to verify that the **mult** and **invert** functions give the identity matrix within the accuracy of the double-precision implementation (about 1.e-16 in the example shown in Listing 6.10).

2. Crammer's rule test—the same 3×3 matrix equation (Equation 6.2) is solved using Crammer's rule using a series of calls to the **det** function. In the MATTST program the solution is calculated by finding the determinant of the transpose of the augmented $\mathbf{A}$ matrix. This does not change the solution obtained by Crammer's rule, because the determinant of $\mathbf{A}^T$ is the same as the determinant of **A.** By using the transpose, however, instead of replacing the columns of the $\mathbf{A}$ matrix to form the augmented matrix determinant as is usually done, the rows of the $\mathbf{A}$ matrix can be replaced by assigning one of the rows to $\mathbf{b}$ for each element of the $\mathbf{x}$ solution vector. The following calculations are performed:

$$x_0 = \frac{\begin{vmatrix} 5 & 6 & 9 \\ 2 & 2 & 2 \\ 1 & 1 & 3 \end{vmatrix}}{\begin{vmatrix} -1 & -2 & -1 \\ 2 & 2 & 2 \\ 1 & 1 & 3 \end{vmatrix}} \quad x_1 = \frac{\begin{vmatrix} -1 & -2 & -1 \\ 5 & 6 & 9 \\ 1 & 1 & 3 \end{vmatrix}}{\begin{vmatrix} -1 & -2 & -1 \\ 2 & 2 & 2 \\ 1 & 1 & 3 \end{vmatrix}} \quad x_2 = \frac{\begin{vmatrix} -1 & -2 & -1 \\ 2 & 2 & 2 \\ 5 & 6 & 9 \end{vmatrix}}{\begin{vmatrix} -1 & -2 & -1 \\ 2 & 2 & 2 \\ 1 & 1 & 3 \end{vmatrix}} \tag{6.3}$$

3. Hilbert matrix test—The Hilbert matrix is a well-known ill-conditioned matrix with an exact inverse. The MATTST program calculates the exact Hilbert mat ix inverse and

compares it to the result obtained using **invert** for Hilbert matrix sizes from 2 to 6. The N by N Hilbert matrix is defined as follows:

$$a_{ij} = \frac{1}{i + j + 1} \tag{6.4}$$

where i and j go from 0 to N - 1. The inverse of this matrix is an integer matrix and is calculated using the iterative algorithm shown in Listing 6.9. The results of this iterative algorithm become very large for even small N (for $N = 5$ the maximum element is 179,200) such that the results cannot be stored in a **short int** matrix. This is an indication of the precision required to invert a Hilbert matrix as is the determinant of the Hilbert matrix (also shown in Listing 6.10). The difference between the exact inverse and the calculated inverse is calculated using the **sub** function. The magnitude of elements of this difference matrix increases with the size of the Hilbert matrix.

6.3 MATRIX DISK STORAGE

The functions **write** and **read** can be used to save and restore the contents of a **Matrix** data structure in the DSP disk data format (see Chapter 3, Section 3.2). Both of these matrix functions are contained in the file DISK.H. The matrix **read** function (shown in Listing 6.11) uses the DSP data format functions to read each record of the DSP data file and requires only the filename of the DSP data file. Each record of the DSP data file becomess a row of the resulting matrix such that the entire DSP data file is read into a single **Matrix** structure. After all the records are stored in the matrix, a pointer to the new **Matrix** structure is returned.

```
//////////////////////////////////////////////////////////////
//
// read( Matrix<Type>& mat )
//   Reads matrix from file. Converts from C++ DSP_FILE type
//   to type of matrix.
//
//  Throws:
//   DSPException
//
//////////////////////////////////////////////////////////////
template <class Type>
void read( Matrix<Type>& mat )
{
```

LISTING 6.11 Function **read** used to read a DSP data file into a **Matrix** structure (contained in file MATRIX.H). (*Continued*)

```
    // Validate state of file
    if( m_fs.is_open() == false )
        throw DSPFileException( "Not opened" );

    if( m_readOnly == false )
        throw DSPFileException( "Not opened for reading" );

    // See to beginning of data
    seek();

    // Allocate matrix for data
    mat.empty();
    mat.init( m_numRecords, m_recLen );

    if( m_type == convType( typeid( Type ) ) )
    {
        for( int i = 0; i < m_numRecords; i++ )
        {
            // Read data directly into row buffer without conversion
          m_fs.read( (BYTE *)mat.m_data[i], sizeof( Type ) * m_recLen );
            if( m_fs.fail() )
                throw DSPFileException( "Reading matrix" );
        }
    }
    else
    {
        // Read data into temporary buffer then convert
        BYTE *rowData = new BYTE[ m_elementSize * m_recLen ];
        if( rowData == NULL )
            throw DSPMemoryException();

        for( int i = 0; i < m_numRecords; i++ )
        {
            // Read row in
            m_fs.read( rowData, m_elementSize * m_recLen );
            if( m_fs.fail() )
            {
                delete [] rowData;
                throw DSPFileException( "Reading matrix" );
            }
            // Convert it to matrix type
            try
            {
                convBuffer( mat.m_data[i], rowData, m_type, m_recLen );
            }
```

LISTING 6.11 (*Continued*)

```
            catch( DSPException& e )
            {
                delete [] rowData;
                throw e;
            }
        }
        delete [] rowData;
    }
}
```

LISTING 6.11 (*Continued*)

The function **write** works in a similar fashion to the **read** function but, no conversion is required because all three **Matrix** structure data types are supported by the DSP data format. The **write** function takes two arguments, the **Matrix** structure name and a character string giving the name of the DSP data file to create. Unlike the **read** function, **write** returns a pointer to the **DSPFile** structure so that further manipulation of the DSP data file can be performed. The most common use of this **DSPFile** structure pointer is to add a descriptive trailer to the matrix data file. For example, suppose that the matrix **A** must be stored on disk to make room for some other large matrix operations. Then latter in the program the **A** matrix can be restored using **read.** The following code segment illustrates **write:**

```
////////////////////////////////////////////////////////////////////////
//
// write( const Matrix<Type>& mat )
//   Writes Matrix to file. Overwrites previous data
//   in file if any. File will be the same type
//   as the matrix.
//
//  Throws:
//   DSPException
//
////////////////////////////////////////////////////////////////////////
template <class Type>
void write( const Matrix<Type>& mat )
{
    // Validate state of file
    if( m_fs.is_open() == false )
        throw DSPFileException( "Not opened" );

    if( m_readOnly )
        throw DSPFileException( "Not opened for writing" );

    // USHRT_MAX is the maximum size of unsigned short
```

```
    // which is used for number of records and record length
    // (USHRT_MAX is defined in limits.h)
    if( mat.rows() > USHRT_MAX || mat.cols() > USHRT_MAX )
        throw DSPFileException( "Matrix too large to write" );

    // See to beginning of data
    seek();

    // File information information
    m_type = convType( typeid( Type ) );
    m_elementSize = sizeof( Type );
    m_numRecords = mat.rows();
    m_recLen = mat.cols();

    // Write out data row by row
    for( int i = 0; i < mat.rows(); i++ )
    {
        m_fs.write( (BYTE *)mat.m_data[i], m_elementSize * m_recLen );
       if( m_fs.fail() )
            throw DSPFileException( "Writing matrix" );
    }
  }
};
```

The **`read`** function can be used to display the contents of a DSP data file by using the following short program (file MATPRINT.CPP):

```
//////////////////////////////////////////////////////////////////
//
// matprint.cpp - Print Matrix
//
//////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
//////////////////////////////////////////////////////////////////
//
// int main()
//   Read and print a matrix
//
//  Returns:
//   0 -- Success
//
//////////////////////////////////////////////////////////////////
int main()
{
    try
```

```
    {
        DSPFile dspfile;
        String strName;
        String strTrailer;

        // Open the input file
        do getInput( "Enter input file name", strName );
        while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        // Get data
        Matrix<float> x;
        dspfile.read( x );

        dspfile.getTrailer( strTrailer );
        dspfile.close();

        cout << strTrailer << endl;

        // Print matrix
        cout << "File records contain the following matrix:\n";
        cout << x << endl;

    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }

    return 0;
}
```

The above program can be used to print out all the elements in any DSP data file. Other examples of the use of the **read** and **write** functions are provided in the rest of this chapter and in Chapter 7.

6.4 LEAST SQUARES CURVE FITTING

This section describes the use of the matrix routines presented in Sections 6.2 and 6.3 for weighted least squares curve fitting. Given a set of (x_i, y_i) data points that form a function (i.e., for each x_i there is only one y_i value), a polynominal of a given order can be fit to these data points by minimizing the following mean squared error expression:

$$E = \sum_{i=0}^{N-1} W_i^2 \left[b_0 + b_1 x_i + b_2 x_i^2 + \ldots + b_L x_i^L - y_i \right]^2. \tag{6.5}$$

where:

W_i = weight value associated with the (x_i, y_i) point
N = number of data points (x_i, y_i) for $i = 0$ to N-1
L = order of the polynomial fit
b_k = desired polynomial coefficients (0 to L)
E = total mean squared error

The weight values (W_i) provide a way to weight the squared error for each data value. Because W_i^2 is used in Equation 6.5, the W_i values should be set so that they are proportional to the inverse of the standard deviation of the y_i measurements (if the y_i data values can be considered to come from a normal distribution). Thus, if a particular sample is noisier than another, the noisy sample contributes less to the mean squared error, which will be minimized. Equivalently, Equation 6.5 can be written in a more compact form as follows:

$$E = \sum_{i=0}^{N-1} W_i^2 \left[\left(\sum_{k=0}^{L} b_k x_i^k \right) - y_i \right]^2 \tag{6.6}$$

By setting the partial derivative of E with respect to b_j for j from 0 to L the minimum E solution is found for the b_j polynomial coefficients with the particular data points. The partial derivatives of E in Equation 6.6 are as follows:

$$\frac{\delta E}{\delta b_j} = 2 \sum_{i=0}^{N-1} W_i^2 \left[\left(\sum_{k=0}^{L} b_k x_i^k \right) - y_i \right] x_i^j \tag{6.7}$$

All $L + 1$ derivatives (for $j = 0$ to L in Equation 6.7) must be set to zero simultaneously. This results in the following equation, which must be solved for b_k:

$$\sum_{k=0}^{L} \sum_{i=0}^{N-1} b_k W_i^2 x_{i+j}^k = \sum_{i=0}^{N-1} W_i^2 x_i^j y_i \tag{6.8}$$

The $L + 1$ equations represented by Equation 6.8 are sometimes called the *normal equations*. They can be solved to determine the least squares curve-fitting solution and are written in matrix notation as follows:

$$\begin{bmatrix} \sum_{i=0}^{N-1} W_i^2 & \sum_{i=0}^{N-1} W_i^2 x_i & \sum_{i=0}^{N-1} W_i^2 x_i^2 & \cdots & \sum_{i=0}^{N-1} W_i^2 x_i^L \\ \sum_{i=0}^{N-1} W_i^2 x_i & \sum_{i=0}^{N-1} W_i^2 x_i^2 & \sum_{i=0}^{N-1} W_i^2 x_i^3 & \cdots & \sum_{i=0}^{N-1} W_i^2 x_i^{L+1} \\ \sum_{i=0}^{N-1} W_i^2 x_i^2 & \sum_{i=0}^{N-1} W_i^2 x_i^3 & \sum_{i=0}^{N-1} W_i^2 x_i^4 & \cdots & \sum_{i=0}^{N-1} W_i^2 x_i^{L+2} \\ .. & .. & .. & \cdots & .. \\ \sum_{i=0}^{N-1} W_i^2 x_i^L & \sum_{i=0}^{N-1} W_i^2 x_i^{L+1} & \sum_{i=0}^{N-1} W_i^2 x_i^{L+2} & \cdots & \sum_{i=0}^{N-1} W_i^2 x_i^{2L} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ .. \\ b_L \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{N-1} W_i^2 y_i \\ \sum_{i=0}^{N-1} W_i^2 x_i y_i \\ \sum_{i=0}^{N-1} W_i^2 x_i^2 y_i \\ \cdots \\ \sum_{i=0}^{N-1} W_i^2 x_i^L y_i \end{bmatrix} \tag{6.9}$$

The preceding solution to the least squares curve-fitting problem is straightforward and does not use linear algebra to form the normal equations. Matrix notation can be used throughout the derivation resulting in a very compact solution to the least squares problem. First, define an N by $L + 1$ matrix $\mathbf{A}$ with elements defined by

$$a_{ij} - W_i x_i^j \tag{6.10}$$

where j is the power of x_i and W_i is the weight associated with y_i as defined previously. The $\mathbf{A}$ matrix defined by Equation 6.10 can then be used to express the least squares minimization problem as follows:

$$E = \| \mathbf{z} - \mathbf{A}\mathbf{b} \|_2^2 \tag{6.11}$$

where $\mathbf{z}$ is an N element vector of weighted observations with each element defined by

$$z_i = W_i y_i \tag{6.12}$$

The vector $\mathbf{b}$ in Equation 6.11 is the vector of $L + 1$ unknown least squares curve fit parameters as was used in Equation 6.9. The symbol $\| \ \|_2^2$ in Equation 6.11 denotes the Euclidean 2-norm of the vector inside which in this case makes E equal to the sum of squares of vector elements. Thus, this formulation of the least squares problem is identical to Equation 6.6. The norm can be expanded as follows:

$$E = \mathbf{z}^{\mathrm{T}}\mathbf{z} - 2\mathbf{z}^{\mathrm{T}}\mathbf{A}\,\mathbf{b} + \mathbf{b}^{\mathrm{T}}\,\mathbf{A}^{\mathrm{T}}\mathbf{A}\,\mathbf{b} \tag{6.13}$$

The derivative of this expression with respect to b can now be determined and set to zero resulting in the normal equations as follows:

$$(\mathbf{A}^{\mathrm{T}}\mathbf{A})\mathbf{b} = \mathbf{A}^{\mathrm{T}}\mathbf{z} \tag{6.14}$$

This equation is a matrix form of Equation 6.9, and because it uses only matrix operations, it is much more compact and easier to solve for least squares parameters ($\mathbf{b}$) as follows:

$$\mathbf{b} = (\mathbf{A}^{\mathrm{T}}\mathbf{A})^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{z} \tag{6.15}$$

Unfortunately, the square $\mathbf{A}^T\mathbf{A}$ matrix is badly conditioned in most cases and is difficult to invert accurately. However, using the matrix functions described in Section 6.2 with double precision, the least squares solution is quite accurate for L less than about 10. Since most applications of curve fitting rarely use an order this large, the implementation of Equation 6.15 is quite practical.

6.4.1 Least Squares Routine

Program LSFIT shown in Listing 6.12 performs least squares curve fitting according to Equation 6.15 using the matrix functions described in Sections 6.2 and 6.3. The program first reads a DSP data file with two or three records representing the least squares data points and an optional weighting vector. The first record is the independent variable (**x[i]**), and the second record is the dependent variable (**y[i]**). If a third record is present in the DSP data file, it is used as the weighting vector (**w[i]**); if only two records are in the DSP data file, then the weighting vector is set to unity for all data points.

Based on the order entered by the user (stored in the variable **1**), the weighted **A** matrix is formed using **w[i]** and **x[i]** for the **n** data points (**i** from 0 to **n - 1**). Because Equation 6.15 requires the transpose of **A** twice and **A** only once, the transpose of **A** (a **L** + 1 by **n** matrix) is created using the following statements:

```
for(i = 0 ; i < n ; i++) {
    at[0][i] = w[i];
    for(j = 1 ; j < (L + 1) ; j++)
        at[j][i] = at[j - 1][i] * x;
```

This creates the increasing powers of x[i] based on the previously calculated row of the **At** matrix. The **Z** matrix is then allocated as an **n** by 1 column vector and is set to **w[i]*y[i]** as defined in Equation 6.12. Equation 6.15 is then implemented by the following two statements:

```
At_Ai_At = mult(invert(mult(At,transpose(At))),At);
B = mult(At_Ai_At,Z);
```

The **Matrix** structure **b** now contains the (**L** + 1) polynomial curve fit parameters. These values are printed by the LSFIT program using **print**. In the last section of the LSFIT program, the **y** values from the least squares estimate and the difference between the least squares estimates and the original y input values are calculated. Both of these vectors are written to a DSP data file with two **n** element records. This output file can be used to determine how well the least squares curve fit parameters match the input data.

6.4.2 Curve-Fitting Examples

This section gives several curve-fitting examples that illustrate the basic operation of the LSFIT program described in the last section. The file LS.DAT on the accompanying disk

```
////////////////////////////////////////////////////////////////////////
//
// lsfit.cpp - Calculate least squares fit
//
////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
////////////////////////////////////////////////////////////////////////
//
// int main()
//   Weighted least squares fit example use of matrix
//   functions.
//   Prints solutions of least squares fit and writes
//   DSPFile with Y best fit and Y error.
//
//  Returns:
//   0 -- Success
//
////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;
        String strTrailer;

        // Open the input file
        do getInput( "Enter x, y, w input file name", strName );
        while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        // Get XYW data
        Vector<float> x;
        Vector<float> y;
        Vector<float> w;
        dspfile.read( x );
        int n = x.length();
        dspfile.read( y );
        if( dspfile.getNumRecords() > 2 )
        {
            // If weights are present use them
            dspfile.read( w );
        }
```

LISTING 6.12 Program LSFIT.CPP used to perform least squares curve fitting on a pair of input data records. (*Continued*)

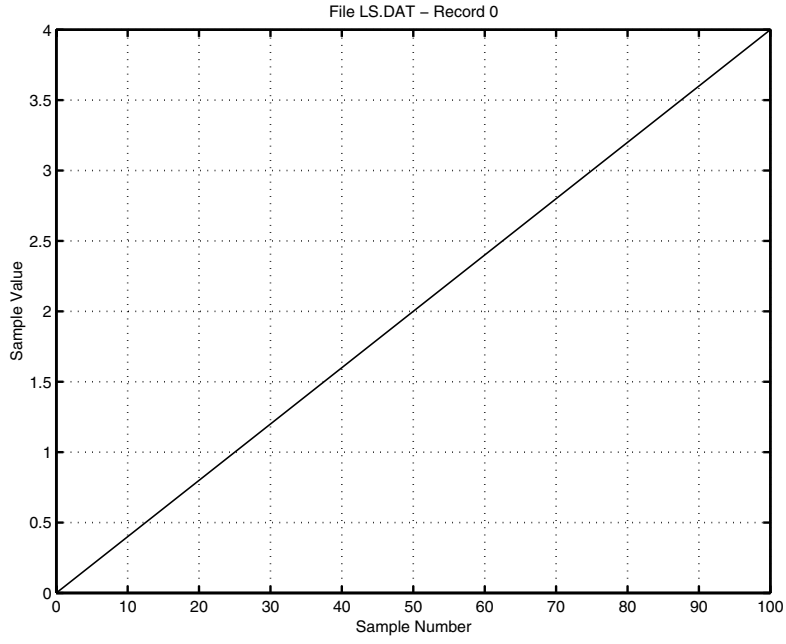
```
        else
        {
            // Otherwise constant weights
            w.setLength( n ) = 1.0f;
        }
        dspfile.getTrailer( strTrailer );
        dspfile.close();

        // Get order of equation
        int order = 0;
        getInput( "Enter order of fit", order, 1, 20 );

        // Weighted powers of x matrix transpose = At
        Matrix<double> At( order + 1, n );
        for( int i = 0; i < n; i++ )
        {
            At[0][i] = w[i];
            for( int j = 1; j < ( order + 1 ); j++ )
            {
                At[j][i] = At[j-1][i] * x[i];
            }
        }

        // Z = weighted y vector
        Matrix<double> z( n, 1 );

        for( i = 0; i < n; i++ )
            z[i][0] = w[i] * y[i];

        Matrix<double> matAtAiAt =
            mult( invert( mult( At, transpose( At ) ) ), At );
        Matrix<double> b = mult( matAtAiAt, z );

        // Print results
        cout << "Least squares solution column vector:\n";
        cout << b << endl;

        // Make a matrix with the fit y and the difference
        Matrix<double> matOut( 2, n );

        // Calculate the least squares y values
        for( i = 0; i < n; i++ )
        {
            double yfit = b[0][0];
            double xpow = x[i];
            for( int j = 1; j <= order; j++ )
            {
```

LISTING 6.12 (*Continued*)

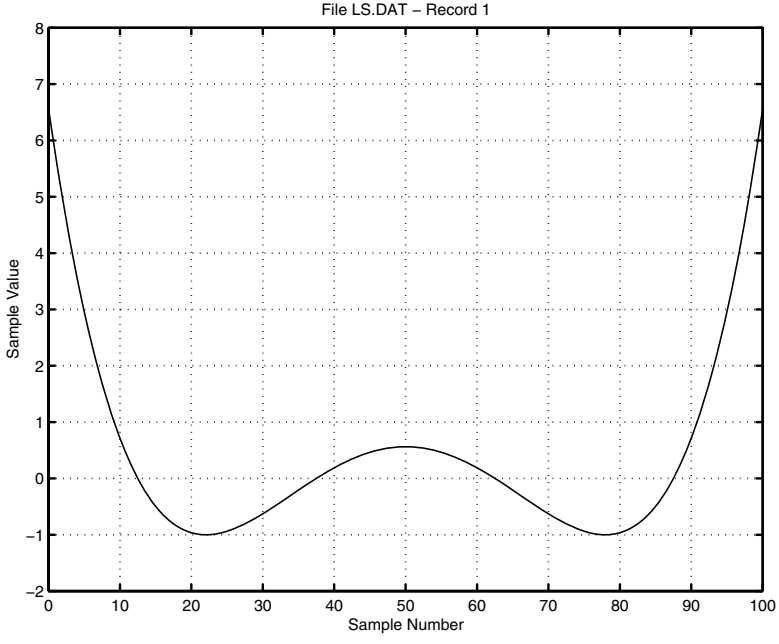
```
                yfit += b[j][0] * xpow;
                xpow *= x[i];
            }
            matOut[0][i] = yfit;
            matOut[1][i] = y[i] - yfit;
        }

        // Write the matrix out
        do getInput( "Enter y, fit, error file name", strName );
        while( strName.isEmpty() );

        dspfile.openWrite( strName );
        dspfile.write( matOut );

        // Make descriptive trailer and write to file
        char fmtBuf[80];
        sprintf(
            fmtBuf,
            "\nLSFIT.CPP program, Order = %d\ny fit and y error\n",
            order );

        // Append to trailer
        strTrailer += fmtBuf;
        dspfile.setTrailer( strTrailer );

        cout << strTrailer << endl;

        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }

    return 0;
}
```

LISTING 6.12 (*Continued*)

is a simple input data file for the LSFIT program. The two 100-point records of LS.DAT are shown in Figure 6.5. The independent variable (**x[i]**), stored in record 0, goes from 0 to 4.0 in a linear fashion. Thus, the dependent data values (**y[i]**), stored in record 1, were "observed" at evenly spaced intervals. Evenly spaced intervals are not required, but it makes the graphic examples presented here easier to understand. The following computer dialogue illustrates the use of the LSFIT program with the LS.DAT data file:



(a)



(b)

FIGURE 6.5 Example least squares curve fit data without noise (file LS.DAT on the accompanying disk). (a) Record 0—independent variable (x). (b) Record 1—dependent variable (y).

```
Enter x,y,w input file name : LS.DAT

Enter order of fit [1...20] : 4

Least squares solution column vector:

Row 0: 6.5625
Row 1: -22
Row 2: 21.5
Row 3: -8
Row 4: 1
Enter y, fit, error file name : LSOUT.DAT
```

In this case, the least squares solution gives an exact representation of the original data because the LS.DAT data was generated using the following fourth-order polynomial:

$$y = x^4 - 8x^3 + 21.5x^2 - 22x + 6.5625$$

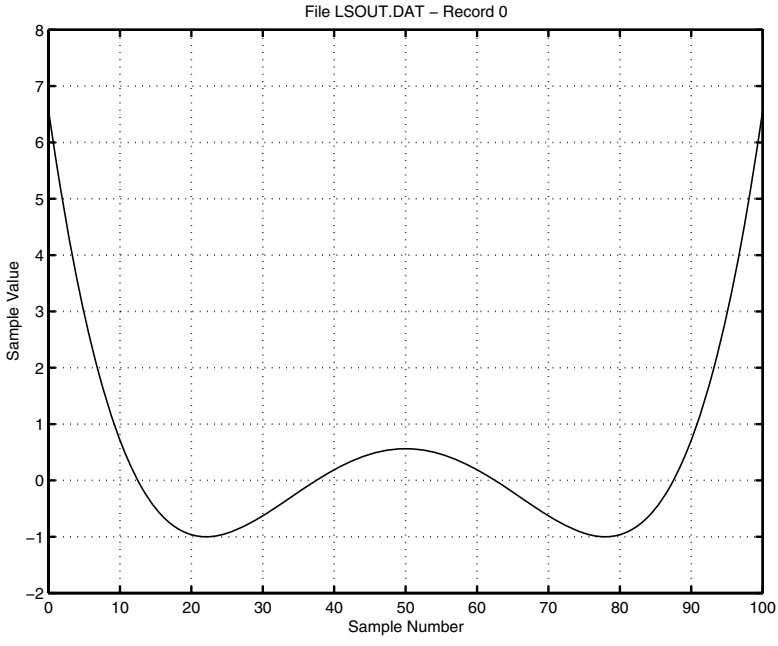
The error in this least squares curve fit can be seen in the LSFIT output file (LSOUT.DAT) shown in Figure 6.6. The first record is the y values obtained using the least squares polynomial coefficients and the second record is the least squares error vector showing the difference between the first record of LSOUT.DAT and the original data. As shown in Figure 6.6(b), the least squares error vector is always less than 2 (10^{-7} and can be attributed to the quantization error associated with the single-precision floating-point format used to store the LS.DAT data file.

If the order of the least squares curve fit of the LS.DAT data file is reduced to 3, the following LSFIT output results:

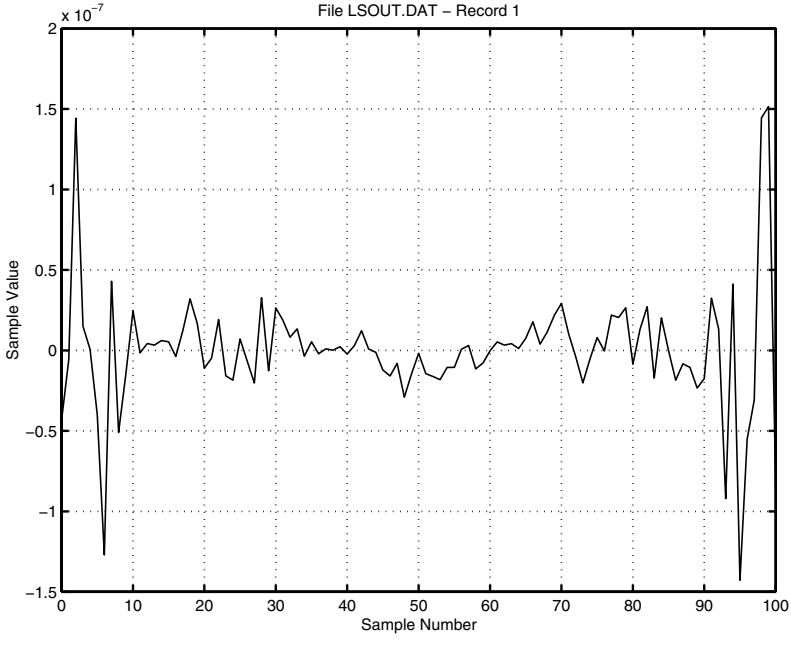
```
Enter x,y,w input file name : LS.DAT

Enter order of fit [1...20] : 3

Least squares solution column vector:
Row 0: 3.12078
Row 1: -3.984
Row 2: 0.996
Row 3: 6.42201e-008
Enter y, fit, error file name : LS3OUT.DAT
```

As shown, the third order least squares coefficients do not agree with the first four fourth order coefficients and the third order term is nearly zero. Figure 6.7 shows the output file (LS3OUT.DAT) from this curve fit which shows the large error obtained with the third order fit. Because the LS.DAT data is symmetrical about its center, the best fit curve of order 3 is actually a second order polynomial. The difference in the least squares values and the original [Figure 6.7(b)] is a fourth-order polynomial. Whenever the least squares error vector indicates a large error with a polynomial shape, the order of the curve fit should probably be increased.

Figure 6.8 shows a version of the fourth-order polynomial data with added Gaussian noise (file LSN.DAT on the accompanying disk) where the noise level in-

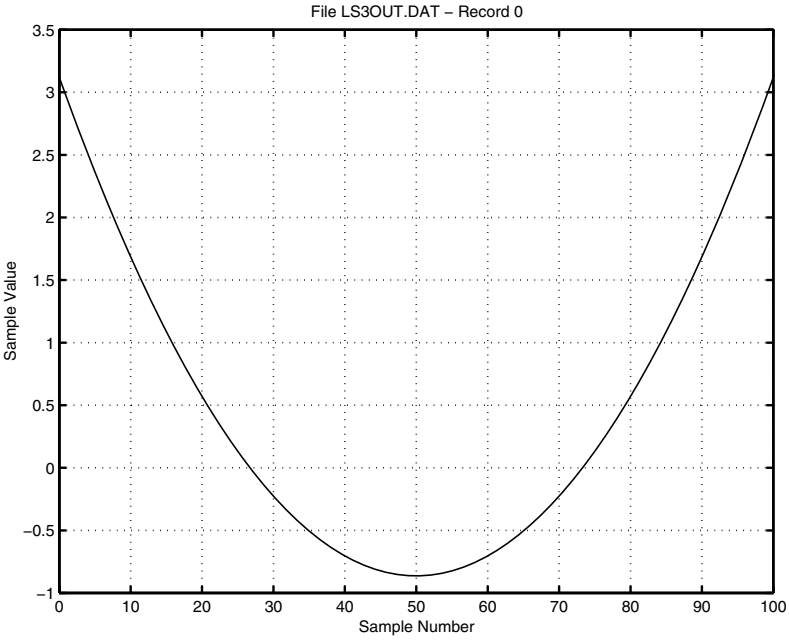


(a)

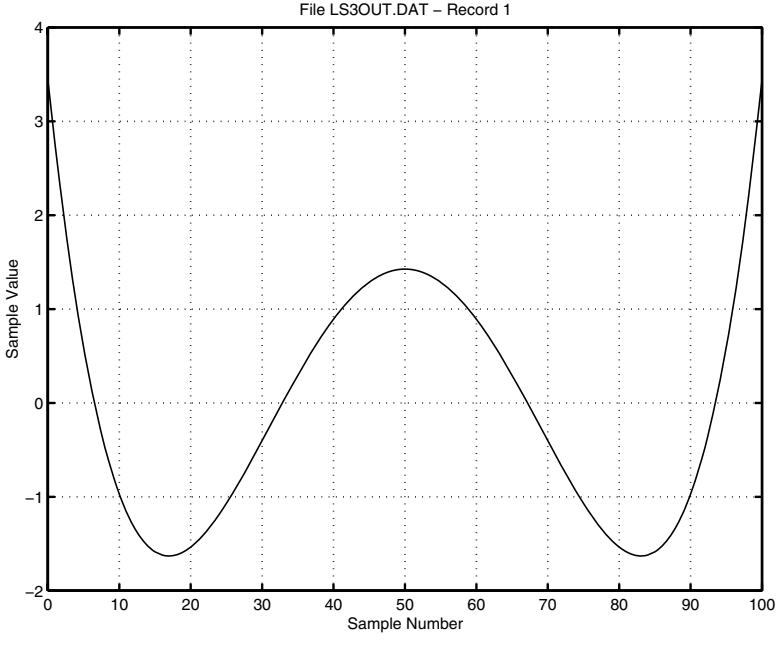


(b)

FIGURE 6.6 Least squares curve fit output from the LSFIT program using the data shown in Figure 6.5. The data was generated using a 4th order polynomial so the error in the 4th order curve fit is negligible. (a) Record 0—Data from fourth-order polynomial fit. (b) Record 1—Error in fourth-order polynomial fit.



(a)



(b)

FIGURE 6.7 Least squares curve fit output using the data shown in Figure 6.5 with a third-order curve fit. (a) Record 0—Data from third-order polynomial fit. (b) Record 1—Error in third-order polynomial fit.

creases with the amplitude of the measurement. For example, this data could represent a series of sensor measurements where the resistance of the sensor was determined by measuring the sensor voltage when a noisy constant current was passed through the sensor. As the sensor resistance increases, the noise voltage also increases. A fourth-order curve fit of this noisy data gives the following results:

```
Enter x,y,w input file name : LSN.DAT

Enter order of fit [1...20] : 4

Least squares solution column vector:
Row 0:     9.90124
Row 1:     -31.10585
Row 2:     29.28356
Row 3:     -10.60735
Row 4:     1.29999
Enter y, fit, error file name : LSNOUT.DAT
```

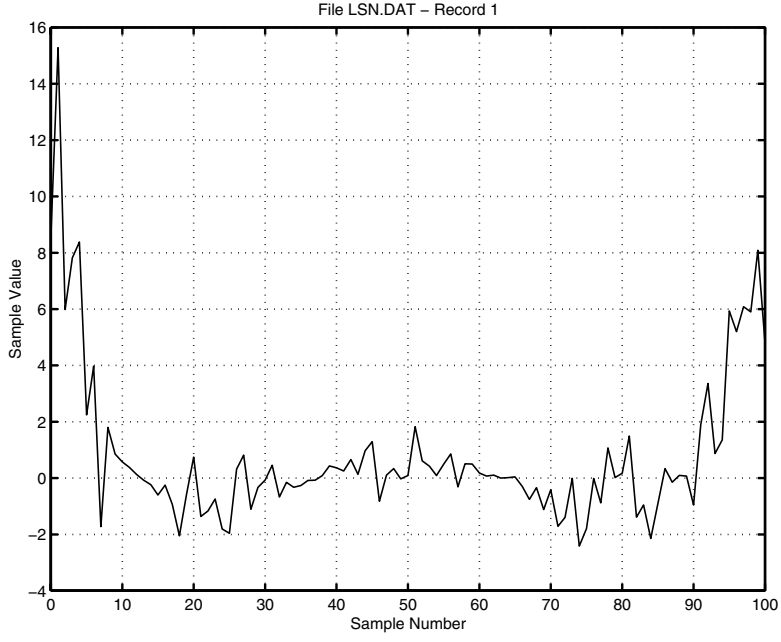


FIGURE 6.8 Least squares curve fit data after being corrupted by adding Gaussian noise to the y data (file LSN.DAT on the accompanying disk). The standard deviation of the noise increased with the magnitude of the samples. [Only record 1 is shown, since record 0 is the same as shown in Figure 6.5(a)].

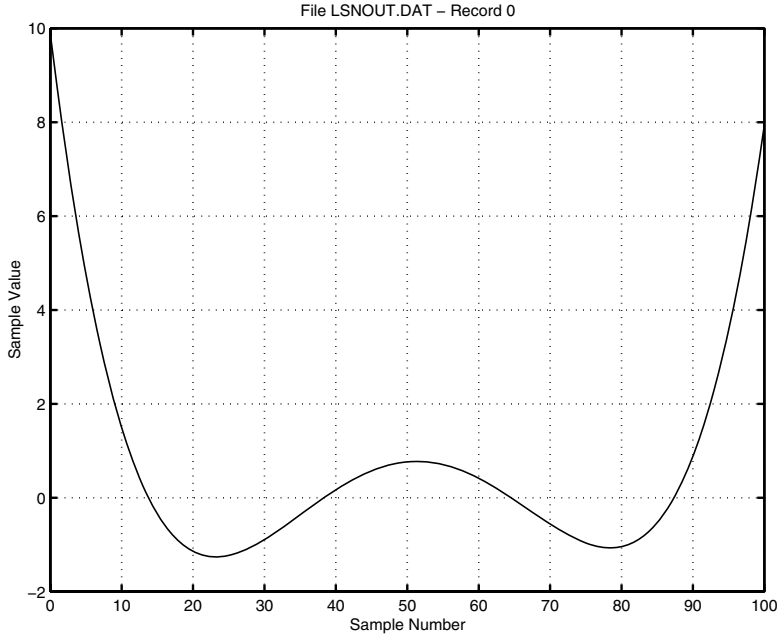
These least squares polynomial coefficients are quite different from the results obtained with the noiseless LS.DAT data file. Figure 6.9 shows the resulting fourth-order fit and the error vector. As shown in Figure 6.9(b), the error vector increases at each end of the data record because the amplitude is largest near the ends, which makes the standard deviation larger. The standard deviation of the noise added to the data was proportional to the absolute value of the **y[i]** data points. In fact, after generating the **y[i]** polynomial data, the Gaussian function (see Chapter 4, Section 4.7.1) was used to add noise using the following statement:

```
y[i] += fabs(y[i]) * gaussian();
```

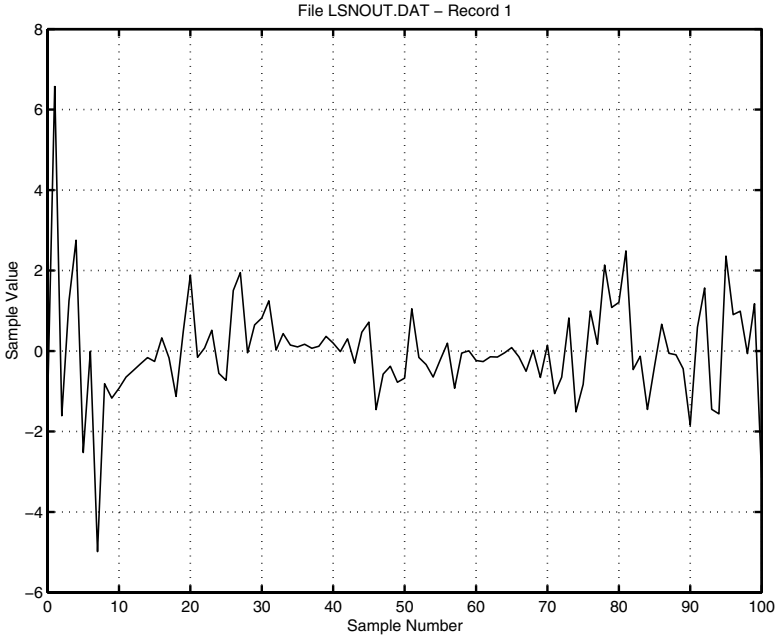
Because the larger amplitude sections of the LSN.DAT data are near each end, a simple weighting function which has more emphasis on the middle of the LSN.DAT data can be used to improve the least squares estimate of the polynomial coefficients. Figure 6.10(a) shows this type of weighting function that is contained in the weighted least squares example data file (record 2 of file LSNW.DAT on the accompanying disk). The first two records of LSNW. DAT are the same as the records in the LSN.DAT data file. The least squares curve fit polynomial coefficients that result when the weighting function in LSNW.DAT is used are much closer to the original noiseless values as the following LSFIT result indicates:

```
Enter x,y,w input file name : LSNW.DAT

Enter order of fit [1...20] : 4

Least squares solution column vector:

Row 0:    6.73328
Row 1:    -22.14216
Row 2:    21.88279
Row 3:    -8.28343
Row 4:    1.05368
Enter y, fit, error file name: LSNWOUT.DAT
```

The least squares error vector of this weighted least squares curve fit (record 1 of file LSNWOUT.DAT) is shown in Figure 6.10(b). Note that the error near each end of the data is larger than the error obtained without weighting [see Figure 6.9(b)]. This is because these data points contribute a very small amount of the total weighted error. The error in 50 samples in the center of the data have a lower error, which accounts for the improved least squares solution.

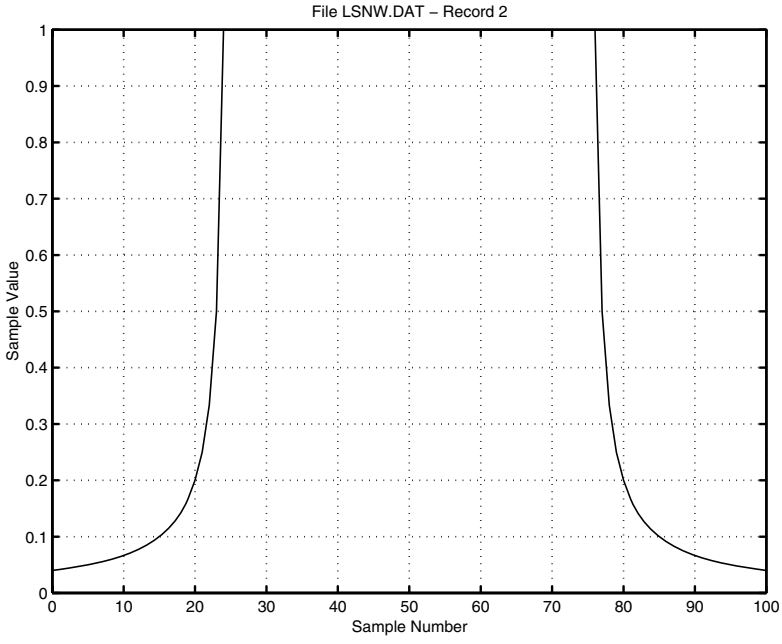


(a)

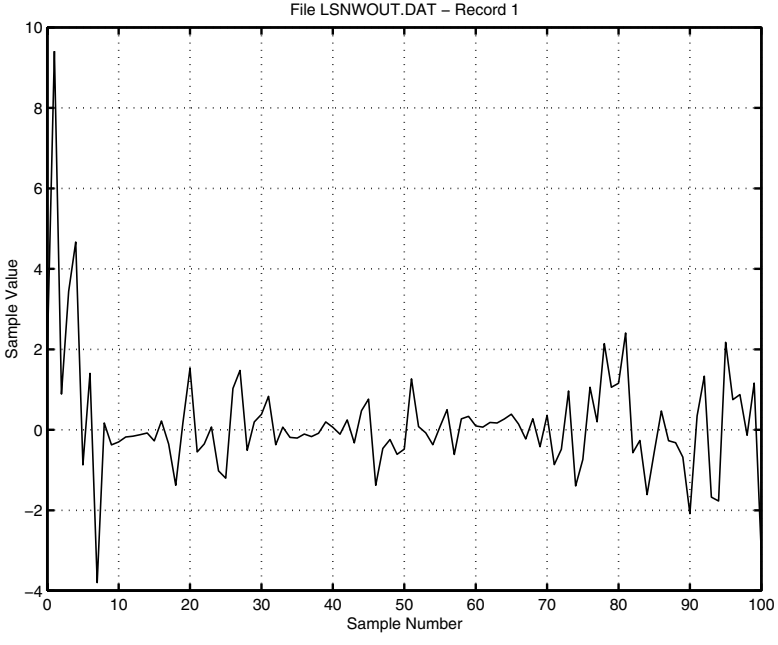


(b)

FIGURE 6.9 Least squares curve fit output from the data shown in Figure 6.8 using a fourth-order curve fit. (a) Record 0—Data from fourth-order polynomial fit. (b) Record 1—Error in fourth-order polynomial fit.



(a)



(b)

FIGURE 6.10 (a) Weighting function of weighted least squares curve fit data (file LSNW.DAT on the accompanying disk). Only record 2 is shown, since record 0 is the same as Figure 6.5(a) and record 1 is the same as Figure 6.8. (b) Least squares error vector of the weighted least squares curve fit (record 1 of file LSNWOUT.DAT).

6.5 EXERCISES

1. Run INTERP (see Chapter 4) to interpolate the radar data file PULSE.DAT by a factor of 3 (use an output file PULSE3.DAT). Run COR to correlate PULSE3.DAT with itself. Use signal offsets of 750 and a correlation length of 600. Use a minimum lag of 0 and a maximum lag of 150. Use PLOT to display the results.
2. Try correlating PULSE3.DAT with a cosine wave generated using MKWAVE. The signal offset for the cosine wave (the X signal) should always be 0. Use a signal offset for the pulse data (the Y signal) of 750, a minimum lag of 0, and a maximum lag of 600. Try a frequency of 0.111 and a length of 100, and then try a frequency of 0.333. Then try a frequency of 0.2. What do these results show?
3. Run COR to correlate CHKL.DAT with itself. Use signal offsets of 4000 and a correlation length of 100. Use a minimum lag of 0 and a maximum lag of 50. Use PLOT to display the results. What does the autocorrelation of speech in this area indicate?
4. Run MATINV (contained on the enclosed disk) to show the inverse of matrix and the product of the inverse and the original. Enter the matrix size (the matrix is square), and then enter the values in the matrix row by row. Try small matrices first (e.g., 2x2) and then try larger ones. Why do some matrices give perfect inverses (which multiply to give the identity matrix) and some don't?
5. Run LSFIT using the (x,y,w) data file LS.DAT and an order equal to 4. Write the error file to disk and display using PLOT. Run LSFIT with an order other than 4. Does the error in the fit improve?
6. Use the ADDNOISE program to add Gaussian noise to the ideal LS.DAT data, run LSFIT again, and observe the results. Use a small multiplier (e.g., 0.01 or use 0.0) for the noise added to record 0 (this is the X input data) and a larger multiplier (e.g., 0.5) for record 1 (this is the Y input data). Run LSFIT on the noisy data with an order other than 4. Does the error in the fit improve?

6.6 REFERENCES

FORSYTHE, G., MALCOLM, M., and MOLER, C., *Computer Methods for Mathematical Computations,* Prentice Hall, Englewood Cliffs, NJ, 1977.

GOLUB, G., and VANCOAN, C., *Matrix Computations*, John Hopkins University Press, Baltimore, 1983.

KAHANER, D., MOLER, C., and NASH, S., *Numerical Methods and Software*, Prentice Hall, Englewood Cliffs, NJ, 1989.

LAWSON, C., and HANSON, R., *Solving Least Squares Problems*, Prentice Hall, Englewood Cliffs, NJ, 1974.

MARPLE, S. L., JR., *Digital Spectral Analysis,* Prentice Hall, Englewood Cliffs, NJ, 1987.

MOLER, C., LITTLE, J., and BANGERT, S., *PC-MATLAB User's Guide,* Math Works, Sherborn, MA, 1987.

PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A., and VETTERLING, W. T., *Numerical Recipes in C,* Cambridge University Press, New York, 1988.

STOER, J., and BULIRSH, R., *Introduction to Numerical Analysis,* Springer-Verlag, New York, 1980.

STRANG, G., *Linear Algebra and Its Applications,* 2nd ed., Academic Press, New York, 1980.

STROUSTRUP, B., *The C++ Programming Language,* Addison-Wesley, Reading, MA, 1986.

WESTLAKE, J. R., *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations*, John Wiley & Sons, New York, 1968.

WILKINSON, J., and REINSCH, C., *Handbook of Automatic Computation*, Vol. 2, *Linear Algebra,* Springer-Verlag, New York, 1971.

CHAPTER 7

Image Processing Routines

In this chapter, the C++ functions developed in previous chapters are applied to the processing of digital representations of image information. The matrix manipulation routines presented in Chapter 6 are used extensively, since all image data will be held in matrix data structures and stored to disk with the matrix input/output functions (see Section 6.3 of Chapter 6). Several new functions are especially useful in image processing and are introduced and used in examples. Examples will be presented for each of the three major areas of image processing.

The first major area of image processing is acquisition, storage, and retrieval of image data. This includes digitization (sampling and quantizing), compression of images, and expansion of compressed images. In Section 7.1 transform techniques are considered. As an example of a transform technique, the *discrete cosine transform* (DCT) is applied to reduce the number of data elements required to transmit or store an image. The inverse transform is then used to retrieve the compressed image. Tradeoffs between block coding size and speed of compression are explored.

The second major area is manipulation and enhancement of individual images. *Histogram equalization* by gray-level modification, *image enhancement, image restoration,* and reconstruction from projections are included in this category. The **histogram** function will be used to modify the distribution of gray levels in an image as an enhancement technique in Section 7.2. Gaussian convolution and median filtering are demonstrated as examples of image noise removal in Sections 7.3 and 7.4, respectively.

The third area of image processing is *comparison* and *abstraction* of images. This category includes image matching and registration, image description and pattern recognition. The examples are in the area of recognition where edge detection for object outlining is demonstrated using two-dimensional convolution in Section 7.3, and object erosion and dilation are shown using nonlinear filtering in Section 7.4.

As in previous chapters, all source code files are on the accompanying disk. Two example 256 x 256 image files are also provided on the disk (BABOON.DAT and LENNA.DAT). The programs in the text are illustrated using the baboon image and the lenna image in order to show different features of the algorithms. The images used by the example programs are stored on disk and retrieved using the matrix disk formats described in Section 6.3 of Chapter 6. Each row in the image is stored in a separate record of the **DSPFile** structure and the elements in the records are successive column values from each row (see Chapter 3 for the DSP data file format). The pixel quantization for the black and white image processing programs and functions in this chapter is 8 bits per pixel, with the gray scale running from 0 (black) to 255 (white). However, to maintain linearity between processes, some of the functions save the images to disk with each element represented by a signed integer (16-bit value). This allows intermediate results to be larger than 255 and less than 0 if required. Any display program used with the outputs of these programs must clip the values at the display limits.

7.1 TRANSFORM TECHNIQUES IN IMAGE PROCESSING

Transforms are used in almost all areas of image processing. A two-dimensional transform is a mathematical transform in both forward and inverse forms. In many image-processing problems (unlike most one-dimensional filtering problems), the transforms need not be exact, because the ultimate performance of the transform is judged by the image observer. Some of the uses of transform techniques in image processing are as follows:

1. Image compression—the image is transformed and only a part of the transform coefficients are retained, the rest being discarded. The image can then be stored most economically and transmitted in a narrower bandwidth or in less time than the original image. When image retrieval is required, the inverse transform is applied and the full image (with some degradation) is recovered (see Rosenfeld).
2. Fast two-dimensional convolution—when the size of the two-dimensional filter used in a convolution is large, the FFT can be used to speed up the convolution. In practice, most convolution kernels are much smaller than the image to be filtered and direct convolution is faster.
3. Reconstruction of images from projections—in ultrasound imaging, synthetic aperture Radar and computer-aided tomography, an image is often reconstructed from

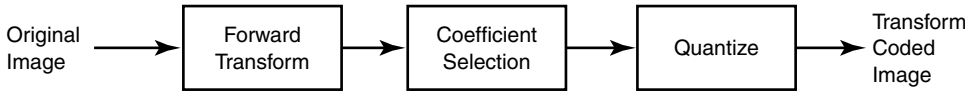


FIGURE 7.1 Transform compression.

many repetitions of a one-dimensional sampled signal. In many cases the FFT is used at several points in the processing (see Rosenfeld).

4. Description of image characteristics—in some machine vision applications, the variation of the coefficients of the two dimensional DFT are used to detect problems in items on an assembly line such as crushed cans or ripped labels. In some military applications a technique called Fourier descriptors has been used to give rotationally invariant descriptions of the outline of objects such as characters of the alphabet, tanks, and airplanes (see Persoon for a description of the technique).

7.1.1 Discrete Cosine Transform Image Compression

Probably the most common use of two-dimensional transforms is image compression. Many transforms have been tried in compression and their performance can be compared by the fidelity of the recovered image to the original (see Pratt, Rosenfeld, or Gonzales and Wintz). The popular JPEG and MPEG compression standards (see Rao) are based on the discrete cosine transform (DCT) similar to the DCT presented in this section. The general flow of the image data in transform compression is shown in Figures 7.1 and 7.2. The sampled image is transformed, a number of the resulting coefficients are chosen from the total, and the remaining coefficients are requantized with fewer bits than the original. For recovery of the image the opposite process is performed: Coefficients are requantized to the original number of bits, missing coefficients are replaced by fixed values (for instance, 0) and the inverse transform is performed.

The fidelity of the process can be measured by taking the difference in intensity level between the original and recovered images at each point in the array. If the original is F_org[m,n] and the recovered is F_rec[m,n] (both N x N matrices), then the difference array is

$$\text{F\_diff}[m,n] = \text{F\_rec}[m,n] - \text{F\_org}[m,n] \tag{7.1}$$

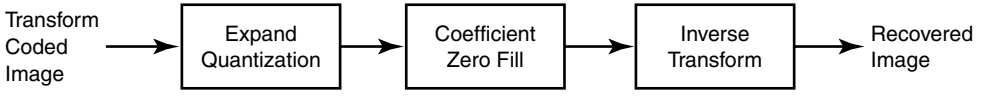


FIGURE 7.2 Recovery of compressed image.

A mean squared error measure of the fidelity can be derived from the difference array:

$$MSE = \frac{1}{N^2} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} \text{F\_diff}[m, n]^2 \tag{7.2}$$

The program MSEOFPIC calculates the mean squared error between two images stored in two DSP data files. This program is included on the disk (file MSEOFPIC.CPP) and is shown in Listing 7.1. The MSEOFPIC program will be used in this section to compare the results of the compression and expansion programs. The two image files are read into **short int** matixes **mRef** and **mTest,** the matrixes are subtracted and the error is accumulated. Note that double precision must be used to accumulate the squared difference to avoid possible overflow of 32 bit integers.

```
////////////////////////////////////////////////////////////////////////
//
// mseofpic.cpp - Mean Squared Difference between two images
//
////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"

////////////////////////////////////////////////////////////////////////
//
// Finds the mean Squared Difference between two
// images in DSPFile format from the user.
//
////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Explain what is going on
        cout << "This program finds the mean squared error of a test\n";
        cout << "image with respect to a reference image.\n";

        // Get reference image
        do getInput( "Enter name of reference matrix", strName );
```

LISTING 7.1 Program MSEOFPIC which reads two images and finds mean squared error. (*Continued*)

```
        while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        // Read matrix from file
        Matrix<short int> mRef;
        dspfile.read( mRef );
        dspfile.close();

        // Get image to test
        do getInput( "Enter name of test matrix", strName );
        while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        // Read matrix from file
        Matrix<short int> mTest;
        dspfile.read( mTest );
        dspfile.close();

        // Check matrices' dimensions
        if( mRef.rows() != mTest.rows() || mRef.cols() != mTest.cols() )
            throw DSPFileException( "Matrix sizes do not match" );

        // Get difference
        double sumErrorSq = 0.0;
        for( int r = 0; r < mRef.rows(); r++ )
        {
            for( int c = 0; c < mRef.cols(); c++ )
            {
                int error = mRef[r][c] - mTest[r][c];
                sumErrorSq += error * error;
            }
        }

      double meanSqError = sumErrorSq/( (double)mRef.rows() * mRef.cols() );

        double rms = sqrt( meanSqError );

        // Format output
        char szBuf[300];
        sprintf(
            szBuf,
            "The mean squared error of the test matrix as compared\n"
            "to the reference matrix is %14.8f.\n"
            "The rms error is         %14.8f.\n",
            meanSqError,
```

LISTING 7.1 (*Continued*)

```
            rms );
        cout << szBuf;
        cout.flush();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
        return 0;
}
```

LISTING 7.1 (*Continued*)

Because the discrete cosine transform gives an MSE result near the theoretical limit of the Karhuenen-Loeve transform (see Rosenfeld), the DCT is very popular for image compression. The definition of the DCT for an N x N image is

$$F[u,\ v] = \frac{1}{N^2}\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} \mathrm{f}\,[m,n]\cos\left[\frac{(2m+1)u\pi}{2N}\right]\cos\left[\frac{(2n+1)v\pi}{2N}\right] \tag{7.3}$$

where:

u,v = discrete frequency variables (0, 1, 2, ..., N - 1).
f[m,n] = N x N image pixels (0, 1, 2, N - 1).
F[u,v] = DCT result.
The inverse DCT is defined as

$$\hat{f}[m,n] = \sum_{m=0}^{N-1}\sum_{n=0}^{N-1} c[u]c[v]\cos\left[\frac{(2m+1)u\pi}{2N}\right]\cos\left[\frac{(2n+1)v\pi}{2N}\right] \tag{7.4}$$

where:

m,n = image result pixel indicies (0, 1, 2, ..., N - 1).
F[u,v] = N x N DCT result to be inverse transformed.
[u] = 1 for g = 0, 2 for g = 1, 2, 3, ..., N - 1.
f[m,n] = N x N inverse DCT result.

7.1.2 Coefficient Quantization in the Compressed Image

The reason image data can be compressed by large factors and successfully recovered with small errors is the large amount of redundancy in typical images. Clearly, if an array of numbers has redundancy, it is theoretically possible to give the same information with less numbers. The purpose of performing the transform is to develop a set of numbers that rep-

resent the image but whose values are uncorrelated (i.e., each number in the array gives new information not given by the other numbers). Since the same information content is to be represented in the original and transformed arrays, some numbers in the transformed array give little or no information about the original image and can be discarded.

The true measure of usefulness of a given coefficient is its variance over a set of images. If a coefficient (say, the coefficient at row 45 and column 99 in a 256 x 256 transformed image) maintains the same value over a set of images, then it does not convey much information about the difference in the pictures in this set. It can, therefore, be replaced with a constant at the receiving end without damaging fidelity. Conversely, if a coefficient has high variance over the set, then it cannot be discarded without serious consequences to the recovered image.

In the cosine transform, variances over typical picture sets tend to have constant variance contours as shown in Figure 7.3. The high-variance coefficients tend to be near the origin and the u and v axes. One way to use this characteristic is to assign a number of quantization levels based on the variance of the coefficient. This is called *zonal coding* (see Pratt) and a typical assignment in a 16 x 16 transformed image is shown in Figure 7.4. This assignment reduces the number of bits required to represent the image from 8 bits per pixel to 1.5 bits per pixel.

7.1.3 Block Coding

When the number of computations for the DFT was considered in Chapter 1, it was shown that the number is of the order of N^2, where N is the length of the transform. The FFT reduces this to order $N \log_2(N)$. Still, the number grows faster than N as the transform length is increased. Two-dimensional transforms have a computation time on the order of N^4, where N is the length of one side of the square image array. Fast transform techniques (using FFTs) can reduce this to order $N^2 \log_2(N)$. In either case, as the length

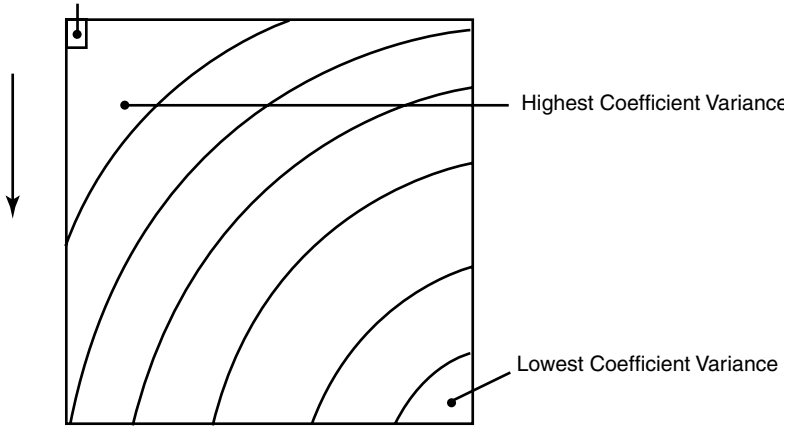


FIGURE 7.3 Lines of constant coefficient variance in typical transformed image.

8	8	8	7	7	7	5	5	4	4	4	4	4	4	4	4
8	8	7	6	5	5	3	3	3	3	3	2	2	2	2	2
8	7	6	4	4	4	3	3	2	2	2	2	2	2	2	2
7	6	4	3	2	2	2	2	1	1	1	1	0	0	0	0
7	5	4	2	2	2	2	1	1	1	0	0	0	0	0	0
7	5	4	2	2	2	1	1	0	0	0	0	0	0	0	0
5	3	3	2	2	1	1	0	0	0	0	0	0	0	0	0
5	3	3	2	1	1	0	0	0	0	0	0	0	0	0	0
4	3	2	1	1	0	0	0	0	0	0	0	0	0	0	0
4	3	2	1	0	0	0	0	0	0	0	0	0	0	0	0
4	3	2	1	0	0	0	0	0	0	0	0	0	0	0	0
4	3	2	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0

FIGURE 7.4 Coefficient bit assignments for transform zonal coding in a 16 x 16 image providing compression from 8 bits per pixel to 1.5 bits per pixel. The upper left corner is the lowest spatial frequency where frequency increases from left to right and top to bottom (adapted from Pratt).

of one side of the image increases, the time required for a two-dimensional transform increases much more rapidly.

In *block coding* this fact is used to reduce the image transform time dramatically. This technique divides an image into many subimages. A transform is then performed on each subimage and the coefficients are quantized just as if each were a separate image. Table 7.1 shows a comparison of transform operations for full-picture coding versus block coding for various block sizes.

However, there is a cost for such a dramatic speed-up in computation. When the block size is large, the degradation due to elimination of coefficients for compression is spread over a large space in the image more or less evenly. As the block size is reduced, the degradation becomes uneven between blocks and the block boundaries show up in the retrieved image. The human vision system is more sensitive to regular patterns of interference such as the block boundaries than to random, scattered interference. As a result, the perceived image quality is less. In addition, because the pixel value correlation in typical images is higher for pixels that are closer to each other in the image, some of the independence of the coefficients is lost. Thus, quantitative measures of image fidelity also degrade.

TABLE 7.1 Comparison of Transform Times for Full Picture (size 256 x 256) and Block Coding. (The transform time for the full 256 x 256 picture is proportional to N^4 = 4.3 x 10^9.)

Block size	Transform Time Factor	Number Required	Total	Speed-up Factor
64 x 64	17 x 10^6	16	3 x 10^8	16
32 x 32	1 x 10^6	64	64 x 10^6	64
16 x 16	65536	256	16 x 10^6	256
8 x 8	4096	1024	4 x 10^6	1024
4 x 4	256	4096	1 x 10^6	4096

To determine the optimum block size one must know the correlation within blocks of adjacent pixels in the original image. In most typical images significant correlation exists for only about 20 adjacent pixels (see Gonzales and Wintz). Therefore, as block size increases above 16 x 16, the increased fidelity is less and less significant. Often, blocks greater than 16 x 16 are not warranted.

7.1.4 Discrete Cosine Transform Functions

Listing 7.2 shows the **dct2d** function, which calculates the DCT as defined in Section 7.1.1. The **dct2d** function first allocates a floating-point matrix for the transformed image storage. The matrix elements that are used repeatedly in the DCT are created the first time the **dct2d** is called and then stored. The actual discrete cosine transform calculations are performed using two matrix multiplies as follows:

$$\mathrm{DCT}\{\mathbf{A}\} = \frac{\mathbf{C}\,\mathbf{A}\,\mathbf{C}^{\mathrm{T}}}{\mathrm{N}^2} \tag{7.5}$$

```
//////////////////////////////////////////////////////////////////////
//
// dct2d( const Matrix<PIXEL>& mIn )
// Performs the Discrete Cosine Transform (DCT) in two dimensions.
//
// Note:
// PIXEL is defined in dsptypes.h
//
// Throws:
// DSPException
//
// Returns:
// Matrix<float> of transformed image.
//
//////////////////////////////////////////////////////////////////////
Matrix<float> dct2d( const Matrix<PIXEL>& mIn )
{
    int N = mIn.rows();
    if( N != mIn.cols() )
        throw DSPMathException( "Non-square DCT input Matrix" );

    if( ( N % DCTBLOCKSIZE ) != 0 )
        throw DSPMathException( "Input matrix doesn't match DCT-block-size" );

    Matrix<float> mCos( DCTBLOCKSIZE, DCTBLOCKSIZE );
```

LISTING 7.2 Function **dct2d** (contained in IMAGE.CPP) used to perform the discrete cosine transform. (*Continued*)

```
    // Fill in the DCT block matrix ( once )
    double factor1 = 2.0 * atan( 1.0 ) / (double)DCTBLOCKSIZE;

    // scale cosine matrix by 1/N for DCT
    for( int u = 0; u < DCTBLOCKSIZE; u++ )
    {
        double factor2 = u * factor1;
        for( int m = 0; m < DCTBLOCKSIZE; m++ )
        {
            mCos[u][m] = (float)cos( ( 2 * m + 1 ) * factor2 );
            mCos[u][m] /= DCTBLOCKSIZE;
        }
    }
    Matrix<float> mCosT = transpose( mCos );
    // Now calculate the DCT by block-matrix multiply
    Matrix<float> mConv;
    conv( mConv, mIn );
    Matrix<float> mOut = transform( mConv, mCos, mCosT );
    return mOut;
}
```

LISTING 7.2 (*Continued*)

Where the **A** matrix is the input matrix (assumed to be square) and the matrix **C** is a matrix of cosine constants with each element defined as follows:

$$c[u][n] = \cos\left[\frac{(2n+1)u\pi}{2N}\right] \tag{7.6}$$

where N is the input matrix size and u and n go from 0 to N - 1. For an 8 x 8 input matrix ($N = 8$), the **C** matrix is as follows:

$$\mathbf{C} = \begin{bmatrix} 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \\ 0.981 & 0.831 & 0.556 & 0.195 & -0.195 & -0.556 & -0.831 & -0.981 \\ 0.924 & 0.383 & -0.383 & -0.924 & -0.924 & -0.383 & 0.383 & 0.924 \\ 0.831 & -0.195 & -0.981 & -0.556 & 0.556 & 0.981 & 0.195 & -0.831 \\ 0.707 & -0.707 & -0.707 & 0.707 & 0.707 & -0.707 & -0.707 & 0.707 \\ 0.556 & -0.981 & 0.195 & 0.831 & -0.831 & -0.195 & 0.981 & -0.556 \\ 0.383 & -0.924 & 0.924 & -0.383 & -0.383 & 0.924 & -0.924 & 0.838 \\ 0.195 & -0.556 & 0.831 & -0.981 & 0.981 & -0.831 & 0.556 & -0.195 \end{bmatrix} \tag{7.7}$$

The scaled **C** and $\mathbf{C}^T$ (each is scaled by $1/N$ in the **dct2d** function) matrices are stored in **Matrix** structures (**mCos** and **mCosT**) for use by calls to the **transform** function which does the block coding and matrix multiplies (see Listing 7.3). After the **float** result matrix is created from the two matrix products, each value is rounded to integer size and placed

in the corresponding pixel location of the input matrix. The **dct2d** function returns a floating point output matrix which is the same size as the integer input matrix.

Listing 7.4 shows the inverse discrete cosine transform function, **idct2d.** The form is similar to **dct2d** but the two matrices (**mCos** and **mCosT**) are reversed in the matrix product step and the special scaling for the inverse transform at $u = 0$ and $v = 0$ is done in the stored matrices. The zero frequency points at $u = 0$ or $v = 0$ (but not both) are multiplied by 2, the point where both u and v equal 0 is not scaled and all other points are scaled by a factor of 4. The final step is the floating point to integer conversion of the output matrix.

```
////////////////////////////////////////////////////////////////////////
//
// Matrix<Type> transform(
//    const Matrix<Type>& m1,
//    const Matrix<Type>& t1,
//    const Matrix<Type>& t2 )
// Transform matrix by multiplying submatrices by transform blocks.
//
// Throws:
// DSPException
//
// Returns:
// Resultant matrix ( t1 * m1 * t2 )
//
////////////////////////////////////////////////////////////////////////
template <class Type>
Matrix<Type> transform(
    const Matrix<Type>& m,
    const Matrix<Type>& t1,
    const Matrix<Type>& t2 )
{
    if( m.isEmpty() || t1.isEmpty() || t2.isEmpty() )
        throw DSPBoundaryException( "Matrix has no data" );

    // Get rows and columns
    int rows = m.rows();
    int cols = m.cols();
    int N = t1.rows();
    if( t1.cols() != N || t2.rows() != N || t2.cols() != N )
        throw DSPMathException( "Non-square transformation matrices" );

    // Input matrix must be divisible by transformation matrix size
    if( ( rows % N ) != 0 || ( cols % N ) != 0 )
        throw DSPMathException( "Transformation matrix over input boundary" );
```

LISTING 7.3 Function **transform** (contained in MATRIX.H) used to perform the block coding for use in the discrete cosine transform function. (*Continued*)

```
    Matrix<Type> mRet( rows, cols );
    Matrix<Type> mtmp( N, N );
    int u, v, i, j, k;
    for( u = 0; u < rows; u += N )
    {
        for( v = 0; v < cols; v += N )
        {
            // Multiply by the t1 matrix
            for( i = 0; i < N; i++ )
            {
                for( j = 0; j < N; j++ )
                {
                    Type tsum = 0;
                    for( k = 0; k < N; k++ )
                        tsum += t1[i][k] * m[u+k][v+j];

                    mtmp[i][j] = tsum;
                }
            }
            // Multiply by t2 matrix
            for( i = 0; i < N; i++ )
            {
                for( j = 0; j < N; j++ )
                {
                    Type tsum = 0;
                    for( k = 0; k < N; k++ )
                        tsum += mtmp[i][k] * t2[k][j];

                    mRet[u+i][v+j] = tsum;
                }
            }
        }
    }
    return mRet;
}
```

LISTING 7.3 (*Continued*)

7.1.5 Image Compression Routine

The COMPRESS program (shown in Listing 7.5) reads an image from a DSP data file and uses the **dct2d** function (see Listing 7.2) to create a compressed version of the image. As with matrices, each row of pixels in the image is contained in a **DSPFile** record. Record number 0 contains the top row in the image, and each value in the record is a pixel amplitude from the successive columns of the row. The entire image is read into a

```
////////////////////////////////////////////////////////////////////
//
// idct2d( const Matrix<float>& mIn )
// Performs the Inverse Discrete Cosine Transform (IDCT)
// in two dimensions.
//
// Note:
// PIXEL is defined in dsptypes.h
//
// Throws:
// DSPException
//
// Returns:
// Matrix<PIXEL> of transformed image.
//
////////////////////////////////////////////////////////////////////
Matrix<PIXEL> idct2d( const Matrix<float>& mIn )
{
    int N = mIn.rows();
    if( N != mIn.cols() )
        throw DSPMathException( "Non-square IDCT input Matrix" );

    if( ( N % DCTBLOCKSIZE ) != 0 )
        throw DSPMathException( "Input matrix doesn't match DCT-block-size" );

    Matrix<float> mCos( DCTBLOCKSIZE, DCTBLOCKSIZE );

    // Fill in the DCT block matrix ( once )
    double factor1 = 2.0 * atan( 1.0 ) / (double)DCTBLOCKSIZE;

    // scale cosine matrix by 1/N for DCT
    for( int u = 0; u < DCTBLOCKSIZE; u++ )
    {
        double factor2 = u * factor1;
        for( int m = 0; m < DCTBLOCKSIZE; m++ )
        {
            if( u > 0 )
                mCos[u][m] = (float)(2.0 * cos( ( 2 * m + 1 ) * factor2 ) );
            else
                mCos[u][m] = (float)cos( ( 2 * m + 1 ) * factor2 );
        }
    }
    Matrix<float> mCosT = transpose( mCos );
```

LISTING 7.4 Function **idct2d** (contained in IMAGE.CPP) used to perform the inverse discrete cosine transform. (*Continued*)

```
    // Now calculate the IDCT by block-matrix multiply
    Matrix<float> mConv = transform( mIn, mCosT, mCos );

    // Round result to the nearest integer
    Matrix<PIXEL> mOut( mConv.rows(), mConv.cols() );
    for( int i = 0; i < mConv.rows(); i++ )
    {
        for( int j = 0; j < mConv.cols(); j++ )
        {
            int temp = round( mConv[i][j] );
            if( temp < 0 )
                temp = 0;
            else if( temp > 255 )
                temp = 255;
            mOut[i][j] = (PIXEL)temp;
        }
    }
    return mOut;
}
```

LISTING 7.4 (*Continued*)

matrix structure using the **read** member function. The compressed version is then written as packed unsigned characters using the **write** member function (see Chapter 3). The size of the compressed output file is a factor of 4 smaller than the input image file (as long as the input image is stored as bytes). This program uses both block coding as discussed in Section 7.1.3 and zonal coding described in Section 7.1.2. The size of the blocks used in the COMPRESS program is 8 x 8, which gives a good compression factor without reducing the efficiency of the algorithm. The 4:1 compression factor is achieved by this simple method by requantizing the frequency component near zero frequency to 8 bits, some frequency values to 4 bits and other values to 0 bits. The pixels are requantized in the following pattern:

```
8   8   4   4   4   4   4   4
8   8   4   4   4   4   0   0
4   4   4   4   0   0   0   0
4   4   4   4   0   0   0   0
4   4   0   0   0   0   0   0
4   4   0   0   0   0   0   0
4   0   0   0   0   0   0   0
4   0   0   0   0   0   0   0
```

```
//////////////////////////////////////////////////////////////////
//
// compress.cpp - Routines to compress images
//
//////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "image.h"
#include "get.h"
//////////////////////////////////////////////////////////////////
//
// Program which uses the Discrete Cosine Transform
// to reduce the amount of data needed to represent
// an image by a factor of 4:1 while maintaining
// maximum information content. A block coding
// technique is used with a block size of 8x8.
//
//////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Open the input file
       do getInput( "Enter image file to be transform coded", strName );
       while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        // Read image into matrix
        Matrix<PIXEL> mIn;
        dspfile.read( mIn );
        dspfile.close();

        int rows = mIn.rows();
        int cols = mIn.cols();

        // Allocate memory for the output records
        Vector<PIXEL> vOut( 2 * cols );

        // Open output PIXEL DSPFile
        do getInput( "Enter compressed output file name", strName );
        while( strName.isEmpty() );
        dspfile.openWrite( strName );
```

LISTING 7.5 Program COMPRESS, which reads an image, finds the DCT of each block, packs the data, and writes the compressed image. (*Continued*)

```
        // Do 8x8 compression, retaining only 8 bit or 4 bits per coef
        Matrix<float> mdct = dct2d( mIn );
        Matrix<int> d( rows, cols );

        // Round floating point numbers to nearest integer
        for( int i = 0; i < rows; i++ )
            for( int j = 0; j < cols; j++ )
                d[i][j] = round( mdct[i][j] );

        // Save image out
        int pos = 0;
        for( i = 0; i < rows; i += DCTBLOCKSIZE, pos = 0 )
        {
            for( int j = 0; j < cols; j += DCTBLOCKSIZE )
            {
                vOut[pos++] = d[i][j];
                vOut[pos++] = d[i][j+1];
                vOut[pos++] = pack( ( d[i][j+2]/3 ), ( d[i][j+3]/3 ) );
                vOut[pos++] = pack( ( d[i][j+4]/2 ), ( d[i][j+5]/2 ) );
                vOut[pos++] = pack( d[i][j+6], d[i][j+7] );
                vOut[pos++] = d[i+1][j];
                vOut[pos++] = d[i+1][j+1];
                vOut[pos++] = pack( ( d[i+1][j+2]/2 ), d[i+1][j+3] );
                vOut[pos++] = pack( ( d[i+1][j+4] ), d[i+1][j+5] );
                vOut[pos++] = pack( ( d[i+2][j]/3 ),( d[i+2][j+1]/2 ) );
                vOut[pos++] = pack( ( d[i+2][j+2]/2 ),( d[i+2][j+3] ) );
                vOut[pos++] = pack( ( d[i+3][j]/3 ),( d[i+3][j+1] ) );
                vOut[pos++] = pack( d[i+3][j+2], d[i+3][j+3] );
                vOut[pos++] = pack( ( d[i+4][j]/2 ), d[i+4][j+1] );
                vOut[pos++] = pack( ( d[i+5][j]/2 ), d[i+5][j+1] );
                vOut[pos++] = pack( d[i+6][j], d[i+7][j] );
            }
            dspfile.write( vOut );
        }
        dspfile.close();
    }
    catch( DSPException& e )
    {
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 7.5 (*Continued*)

Each element in each 8 x 8 submatrix is quantized as shown above and packed into 16 **unsigned chars** (type **Vector<PIXEL>).** The coefficients where a four is shown in the pattern above are requantized to 4 bits (-8 to 7) from their original integer values by the **pack** function (included in file COMPRESS.CPP) which is as follows:

```
PIXEL pack( short int a, short int b )
{
        // Limit to signed 4 bits
    if( a > 7 ) a = 7;
    else if( a < -8 ) a = -8;

    if( b > 7 ) b = 7;
    else if( b < -8 ) b = -8;

    // Pack into 8 bits and return
    return( (PIXEL)( ( a << 4 ) | ( b & 15 ) ) );
}
```

The zero frequency component (at the upper left corner of the above diagram) is quantized to an **unsigned char** since the image data is all positive and this value is the most important term in the DCT output. The above pattern was chosen to make the packing of the coefficients simple, efficient and to maintain the contours of constant variance of the transform coefficients shown in Figure 7.3. In order to avoid clipping in the coefficients which are quantized to only 4 bits, several of the coefficients are divided by 2 or 3 before quantization (see Listing 7.5). This scaling process must be reversed in the expansion program (see Section 7.1.6).

The 28 quantized DCT coefficients are packed into 16 **unsigned chars** by a series of calls to the **pack** function. Thus, because the original 64 pixels required 512 bits of storage, a 4:1 compression ratio is obtained. Each group of 8 rows in the original image is packed into an **unsigned char** array (allocated at the beginning of the COMPRESS program) with a length twice as large as the number of columns in the input image. The **write** function is then used to write each record to the compressed output file. The compressed image size is $N/8$ records by $2N$ bytes in each record, where N is the size of the original square image.

7.1.6 Image Recovery Routine

The EXPAND program (shown in Listing 7.6) reads the compressed image data (created by the COMPRESS program), unpacks the compressed image records, and then writes the reconstructed image to a DSP data file using the **write** function. The EXPAND pro-

gram performs the inverse of the operations performed by the **compress** function. The quantized coefficients are first unpacked by using a series of calls to the **unpack** function and copied into an integer **Matrix** structure after the inverse of the coefficient scaling performed by the COMPRESS program. The inverse discrete cosine transform is then applied using the function **idct2d** (see Listing 7.3) and the resulting image data is written using the **write** function. The **unpack** function (included in file EXPAND.CPP) is as follows:

7.1.7 Compression and Recovery of an Image

Figure 7.5 shows the original image of the baboon (file BABOON.DAT on the accompanying disk) and Figure 7.6 shows the baboon image after compression and recovery using

```
signed char unpack( PIXEL in )
{
    // Only lower 4 bits
    in = (PIXEL)( in & 0xF );
    // Check for negative
    if( in > 0x07 )
        return( (signed char)( in - 0x10 ) );
    else
        return( (signed char)in );
}

//////////////////////////////////////////////////////////////////////
//
// expand.cpp - Routines to expand images
//
//////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "image.h"
#include "get.h"
//////////////////////////////////////////////////////////////////////
//
// Program which uses the Inverse Discrete Cosine
// Transform to demonstrate the operation of expansion
// of an image coded using dct2d and compress.cpp.
// The encoded image is decoded using a 8x8 block
// transform technique and the idct2d function.
//
//////////////////////////////////////////////////////////////////////
```

LISTING 7.6 Program EXPAND, which reads a compressed image, unpacks the compressed data, finds the IDCT of each block, and writes the recovered image. (*Continued*)

```
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Open the input file
        do getInput( "Enter image file to uncompress", strName );
       while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        // Check for correct image type
        if( dspfile.getType() != UNSIGNED_CHAR )
          throw DSPFileException( "Incorect compressed file data type" );

        // Compressed size
        int rowsIn = dspfile.getNumRecords();
        int colsIn = dspfile.getRecLen();
        cout << "Compressed image size: " << rowsIn << " x " << colsIn << endl;

        Vector<PIXEL> vIn;

        // Uncompressed size
        int rowsOut = 8 * rowsIn;
        int colsOut = colsIn / 2;
        Matrix<float> mOut( rowsOut, colsOut );

        // Set all elements to 0
        mOut = 0.0f;

        // Read in the compressed vector
        int pos = 0;
        dspfile.read( vIn );

        // Decode the compressed image
        for( int i = 0; i < rowsOut; i += DCTBLOCKSIZE )
        {
            if( pos == vIn.length() )
            {
                // Read next compressed vector
                pos = 0;
                dspfile.read( vIn );
            }
```

LISTING 7.6 (*Continued*)

```
            for( int j = 0; j < colsOut; j += DCTBLOCKSIZE )
            {
                // Unpack 16 bytes for each 8x8 block
                // DC Value is unsigned
                mOut[i][j] = (unsigned char)vIn[pos++];
                mOut[i][j+1] = (signed char)vIn[pos++];
                mOut[i][j+2] = 3 * unpack( vIn[pos] >> 4 );
                mOut[i][j+3] = 3 * unpack( vIn[pos++] );
                mOut[i][j+4] = 2 * unpack( vIn[pos] >> 4 );
                mOut[i][j+5] = 2 * unpack( vIn[pos++] );
                mOut[i][j+6] = (signed char)unpack( vIn[pos] >> 4 );
                mOut[i][j+7] = (signed char)unpack( vIn[pos++] );

                mOut[i+1][j] = (signed char)vIn[pos++];
                mOut[i+1][j+1] = (signed char)vIn[pos++];
                mOut[i+1][j+2] = 2 * unpack( vIn[pos] >> 4 );
                mOut[i+1][j+3] = unpack( vIn[pos++] );
                mOut[i+1][j+4] = unpack( vIn[pos] >> 4 );
                mOut[i+1][j+5] = unpack( vIn[pos++] );

                mOut[i+2][j] = 3 * unpack( vIn[pos] >> 4 );
                mOut[i+2][j+1] = 2 * unpack( vIn[pos++] );
                mOut[i+2][j+2] = 2 * unpack( vIn[pos] >> 4 );
                mOut[i+2][j+3] = unpack( vIn[pos++] );
                mOut[i+3][j] = 3 * unpack( vIn[pos] >> 4 );
                mOut[i+3][j+1] = unpack( vIn[pos++] );
                mOut[i+3][j+2] = unpack( vIn[pos] >> 4 );
                mOut[i+3][j+3] = unpack( vIn[pos++] );

                mOut[i+4][j] = 2 * unpack( vIn[pos] >> 4 );
                mOut[i+4][j+1] = unpack( vIn[pos++] );
                mOut[i+5][j] = 2 * unpack( vIn[pos] >> 4 );
                mOut[i+5][j+1] = unpack( vIn[pos++] );
                mOut[i+6][j] = unpack( vIn[pos] >> 4 );
                mOut[i+7][j] = unpack( vIn[pos++] );
            }
        }
        dspfile.close();

        // Transform compressed image back
        Matrix<PIXEL> midct = idct2d( mOut );

        // Open output PIXEL DSPFile
        do getInput( "Enter uncompressed output file name", strName );
```

LISTING 7.6 (*Continued*)

```
        while( strName.isEmpty() );
        dspfile.openWrite( strName );
        dspfile.write( midct );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 7.6 (*Continued*)

the COMPRESS and EXPAND programs (file BABEXP.DAT). The COMPRESS and EXPAND programs require only the input and output file names, since the parameters of the compression algorithm are fixed. As shown in Figures 7.5 and 7.6, the recovered image is very similar to the original image. A quantitative measure of the similarity of the two images can be obtained using the MSEOFPIC program, which gives the root mean squared error as follows:

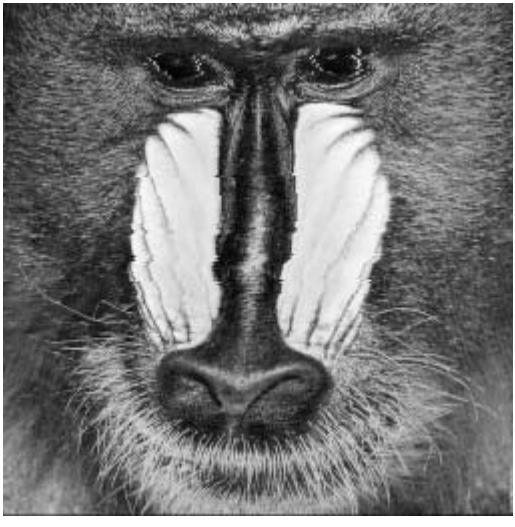


FIGURE 7.5 Original baboon image (File BABOON.DAT).

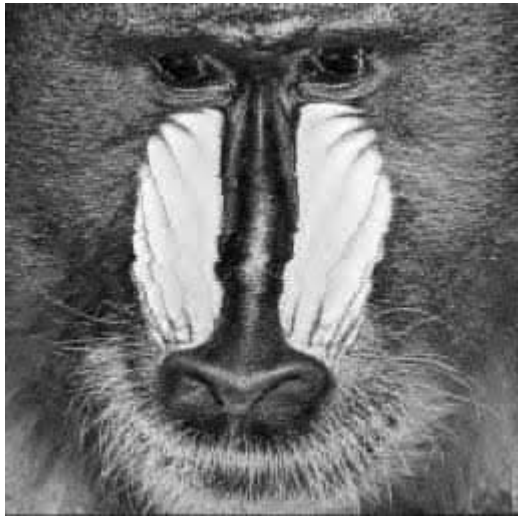


FIGURE 7.6 Baboon image after compression by 4:1 and recovery using programs and COMPRESS and EXPAND.

```
Enter name of reference matrix : BABOON.DAT

Enter name of test matrix: BABEXP.DAT

The mean squared error of the test matrix as compared
to the reference matrix is    185.73716736
The rms error is            13.62854238
```

Figure 7.7 shows the original Lenna image and Figure 7.8 shows the Lenna image after compression and recovery. As with the baboon image, the recovered image is almost the same as the original. The MSEOFPIC program gives the following results:

```
Enter name of reference matrix : LENNA.DAT

Enter name of test matrix : LENEXP.DAT

The mean squared error of the test matrix as compared
to the reference matrix is 79.26679993
The rms error is             8.90319044
```

The root mean squared error for the lenna image is somewhat smaller than the mean squared error for the baboon image primarily because the lenna image has large areas with approximately the same grey level. In these regions, the zero frequency term domi-

FIGURE 7.7 Original image of Lenna.

FIGURE 7.8 Lenna image after compression by 4:1 and recovery using programs COMPRESS and EXPAND.

nates the DCT result, and because this term is quantized with full 8-bit precision, very little information is lost. In the baboon image, where the whiskers and hair contain more high frequency information, some loss of information occurs. However, careful examination of the images reveals that the lenna image has more compression artifacts including a blocky appearance due to the block-coding technique. This simple observation illustrates why image-processing algorithms must be evaluated in a qualitative as well as quantitative manner.

7.2 HISTOGRAM PROCESSING

The histogram of a picture is a function giving the number of pixels at a particular gray level versus the gray level. A histogram can be thought of as a discrete probability density function for an individual image in the following way: Each trial in the probability experiment is the selection of a pixel at random from the image and the event measured in the gray level of the selected pixel. When the probability density function at each gray level is multiplied by the number of pixels in the image, the histogram values are the result. The histograms of the example images are given in Section 7.2.3 where histogram flattening is demonstrated.

The histogram has no information about location of pixels or one pixel's proximity to another. It does convey information about the apparent brightness and contrast of an image and is used in image processing to manipulate these features of an image. Once the histogram of an image is known, the gray levels in the image can be manipulated to change the histogram as desired. The purpose may be to improve contrast, change brightness level, or match the histogram of another picture.

Any gray-level modification technique, of which histogram modification is one example, is based on creating a mapping of the gray levels in the original image to the gray levels in the modified image. Let the gray levels in the original 8-bit image be represented by a variable w, which can take on integer values from 0 to 255. Let the gray levels in the modified image be represented by q with the same range of values. Then the transformation equation is

$$q = T(w)$$

where $T(\ )$ is some transformation operator (the notation is not meant to imply a C function). This means that for a pixel in the original image with the gray level w, the pixel at the same location in the modified image will be given gray level $T(w)$.

In histogram flattening, the desired result is that the probability density of q (the gray levels in the modified image) be uniform for all values of q from 0 to 255. The probability density function can be obtained from the histogram values for the original image. For example, if the gray level w is equal to 120, then the value of $p_w(120)$ is the number of occurrences of the gray level 120 in the image [the histogram of the image at 120 or $h(120)$, where h is the histogram] divided by the total number of pixels in the image or

$$p_w(r) = h(r)/N^2$$

where N^2 is the number of pixels in the image and r is the gray level (r = 120, for example). It can be shown (see Gonzalez and Wintz) that by using the *cumulative distribution function* (CDF) of w (scaled by the maximum gray level) as the values for q, the most uniform probability density is obtained. The equation for the transformation is

$$T(w) = I_{\max} \sum_{r=0}^{w} p_w(r) \tag{7.8}$$

where r is a summation variable and $I_{\max}$ is the maximum gray level (255 for an 8-bit image). The right-hand side of the transformation equation is the sum of the discrete probability densities from $r = 0$ to w (the original image gray level), also called the cumulative distribution function of the variable w. If the histogram of the image can be found, it is clear that the CDF can be found for each original gray level and, from this, the modified gray level is determined.

As a check on the plausibility of this method, assume the original image already has a uniform histogram. This means that there are an equal number of pixels at each individual gray level from 0 to 255. If the picture is a 256 x 256 size image (making N^2 = 65536), the number of pixels at each gray-level will be 256. Thus the value of $p_w(n) = 256/N^2$ for all gray-level values. The histogram-flattened picture will use the CDF at each gray-level value, scaled by $I_{\max}$ (which is 255), as the new gray level. The CDF will be a linear function from 0 to 1.0 causing $T(w)$ to be exactly the same as the original gray levels.

From the discussion above, a strategy for the histogram flattening program can be derived. The first step is to determine the histogram of the original image from which the discrete *probability density function* (PDF) can be derived by a scale factor. By accumulating these PDF values from 0 to each individual gray level, a transformation array is formed whose indices are the original gray levels and whose values are the modified image gray levels.

The next section describes the function **histogram**, which determines the histogram of an image. Sections 7.2.2 and 7.2.3 give an example of the use of **histogram** to create an image with a nearly uniform histogram.

7.2.1 Histogram Function

The function **histogram** (contained in the file IMAGE.CPP) is shown in Listing 7.7. This function accepts a pointer to a **Matrix** structure (assumed to be **PIXEL** type) and two integers specifying the range of histogram gray levels (**min** to **max,** inclusive) as its parameters. For example, for an 8-bit image, the **min** parameter should be 0 and the **max** parameter should be 255 so that the histogram is calculated over the entire range of pixel values. Image gray values outside the specified range are clipped and are counted as values at the maximum or minimum of the range. The function returns an

```
////////////////////////////////////////////////////////////////////
//
// histogram( const Matrix<PIXEL>& mIn, int min, int max )
// Calculates histogram of 2D image.
//
// Note:
// PIXEL is defined in dsptypes.h
//
// Throws:
// DSPException
//
// Returns:
// Vector<float>
//
////////////////////////////////////////////////////////////////////
Vector<float> histogram( const Matrix<PIXEL>& mIn, int min, int max )
{
    if( min > max )
        throw DSPParamException( "Histogram max less than min" );
    Vector<float> hist( max - min + 1 );
    hist = 0;

    // Step through the input matrix. Make sure each value is within
    // the proper range for an index into the histogram array. Then
    // increment the histogram value at this index point
    for( int i = 0; i < mIn.rows(); i++ )
    {
        const PIXEL *inrow = mIn[i];
        for( int j = 0; j < mIn.cols(); j++ )
        {
            PIXEL tempVal = *inrow++;
            if( tempVal > max )
                tempVal = max;
            else if( tempVal < min )
                tempVal = min;
            hist[tempVal - min]++;
        }
    }
    return( hist );
}
```

LISTING 7.7 Function **histogram** used to find the histogram of a **Matrix** structure (contained in file IMAGE.CPP).

array of floating point numbers whose values are the histogram magnitudes for the gray levels selected.

The histogram function proceeds in the following way. The floating-point result **Vector** (**hist**) is allocated and all values are cleared. Stepping through the input matrix, the gray level at each image point is clipped to ensure that it lies within the selected range, and then used as an index into the floating-point histogram array. The value in the histogram at this index is incremented, indicating that one more pixel was found at this particular gray level. When the entire image matrix has been scanned, the function returns the floating-point array.

7.2.2 Histogram-Flattening Routine

The program FLATTEN (shown in Listing 7.8) uses the function **histogram** and the technique described in Section 7.2 to flatten the histogram of an input image. The program prompts the user for an input file which it reads into a **Matrix** structure. It applies the **histogram** function to this input matrix and stores the returned array as the first record in a **DSPFile.** The cumulative distribution function of the input image is created by adding the probability density value (histogram divided by the image size) at each point in the gray level range to the previous CDF total. In the same loop, the gray level mapping is created by multiplying the CDF by 255 to increase its range from 0 to 1.0 to the full gray range of 0 to 255.

An output matrix to hold the histogram flattened image is then allocated and the gray level mapping applied to the matrix **mIn.** The values are placed in matrix **mOut.** After **mOut** is created, the function **histogram** is applied to it and the resulting histogram array is written as the second record of the **DSPFile.** The final operation in the program is to write the matrix **mOut** (the histogram flattened version of **mIn**) to disk in **DSPFile** format.

7.2.3 Histogram-Flattening Example

In this example, the image file BABOON.DAT is histogram flattened using the routine FLATTEN. The computer dialogue is as follows:

```
Enter name of image file to be histogram flattened: BABOON.DAT
Enter histogram file name: BABOON.HST
Enter name of the histogram flattened image: BABFLT.DAT
```

By following the above procedure, the two files BABOON.HST and BABFLT.DAT are created. There are two records in BABOON.HST containing the histograms of the baboon image before and after flattening. The contents of these records are plotted in Figures 7.9(a) and 7.9(b). Figure 7.9(b) is clearly the more uniform histogram. This uniformity has been achieved by combining smaller groups of pixels with adjacent gray levels and by interspersing zero-count gray-level groups throughout the histogram. The re-

```
////////////////////////////////////////////////////////////////////
//
// flatten.cpp - Flatten histogram of image
// Program uses the histogram function to flatten
// the histogram of an input image.
//
// Inputs:
// Histogram of an input image
//
// Outputs:
// Flattened histogram and image
//
////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "image.h"
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Open the input file
        do getInput( "Enter image file to be histogram flattened", strName );
        while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        Matrix<PIXEL> mIn;
        dspfile.read( mIn );
        dspfile.close();

        // Make descriptive trailer for output image and histogram file
        String strHistTrailer;
        strHistTrailer = "Histograms of file ";
        strHistTrailer += strName;
        strHistTrailer += " before and after flattening";

        String strImageTrailer;
        strImageTrailer = "Image ";
        strImageTrailer += strName;
        strImageTrailer += " after flattening";
```

LISTING 7.8 Program FLATTEN used to make the histogram of an image more uniform. (*Continued*)

```
        // Create a vector of representing the number
        // of pixels in the image at a particular gray level.
        Vector<float> histArray = histogram( mIn );

        // Use the histogram vector to create a mapping of the original
        // gray levels to the new histogram flattened gray levels
        double tempFlt = 0.0;

        // Calculate the number of pixels in the image
        double imageSize = mIn.rows() * mIn.cols();

        // Vector to store new gray levels
        Vector<int> newGrayLevel( 256 );
        for( int i = 0; i < 256; i++ )
        {
            // Loop thru original gray levels

            // Find the distribution function of the image at each gray level
            tempFlt += histArray[i] / imageSize;

            // Use the distribution function to create the mapping
            // from old to new gray levels
            newGrayLevel[i] = (int)( ( 255.0 * tempFlt ) + 0.5 );
        }

        // Using the new mapping of gray levels create a new image
        // from the original
        Matrix<PIXEL> mOut( mIn.rows(), mIn.cols() );

        for( i = 0; i < mIn.rows(); i++ )
        {
            for( int j = 0; j < mIn.cols(); j++ )
            {
                int tempVal = mIn[i][j];
                if( tempVal > 255 )
                    tempVal = 255;
                else if( tempVal < 0 )
                    tempVal = 0;

                mOut[i][j] = (PIXEL)newGrayLevel[tempVal];
            }
        }

        do getInput( "Enter histgram file name", strName );
        while( strName.isEmpty() );
```

LISTING 7.8 (*Continued*)

```
        // Write the histograms to disk as two records
        dspfile.openWrite( strName );
        dspfile.write( histogram( mIn ) );
        dspfile.write( histogram( mOut ) );
        dspfile.setTrailer( strHistTrailer );
        dspfile.close();

        // Write new image
        do getInput( "Enter histogram flattened image name", strName );
        while( strName.isEmpty() );

        dspfile.openWrite( strName );
        dspfile.write( mOut );
        dspfile.setTrailer( strImageTrailer );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 7.8 (*Continued*)

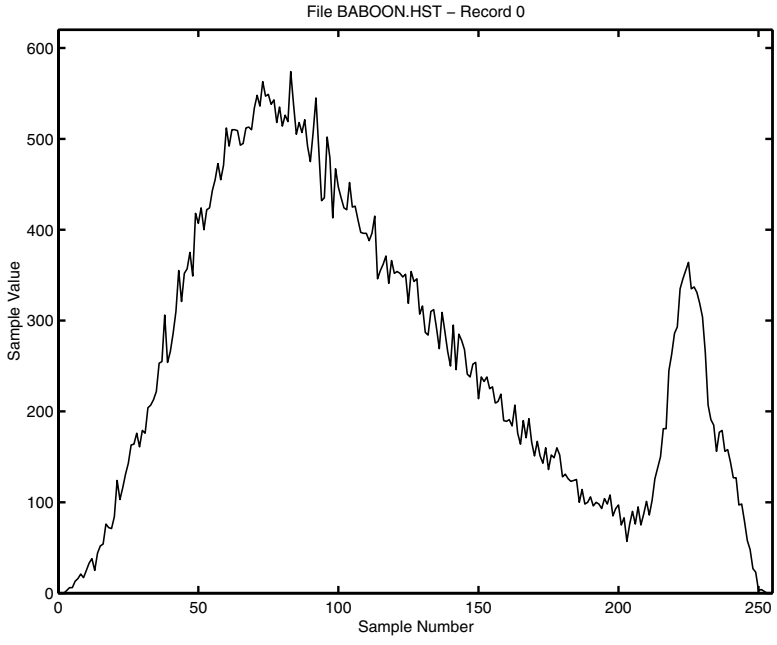
sult is a flatter plot overall, although it is still far from an equal count per gray-level (or uniform) histogram.

Figure 7.10 shows the resulting image of the baboon after the flattening process (file BABOON.FLT in the above dialogue). The increased contrast throughout the picture is very evident when compared with the original in Figure 7.5. The value of this histogram flattening process for pictures with poor exposures or which were taken in poor lighting conditions is clear.

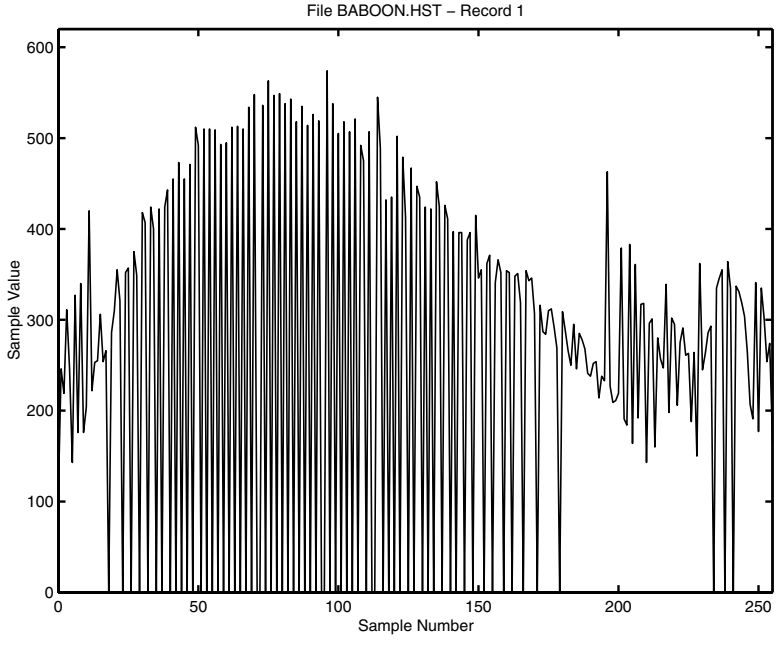
The same FLATTEN program was run on the lenna test image (see Figure 7.7). The histograms are shown in Figure 7.11a and b. The flattened image is shown in Figure 7.12 and can be compared with the original in Figure 7.7. As with the previous baboon image, the histogram is more uniform after flattening and the picture shows much improved contrast.

7.3 TWO-DIMENSIONAL CONVOLUTION

The most common class of processes on images is convolution, which is equivalent to two-dimensional linear filtering. Convolution is used for image sharpening, noise removal, edge detection, and image preprocessing. It can also be used for feature detection



(a)



(b)

FIGURE 7.9 (a) Histogram of the baboon image (BABOON.DAT). (b) Histogram of baboon image after using the FLATTEN program to make the histogram more uniform.

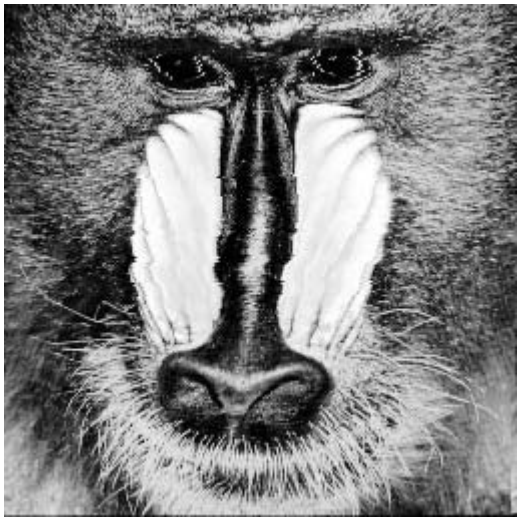


FIGURE 7.10 Baboon image after histogram flattening using program FLATTEN.

(such as lines and corners) and object matching and registration. The equations for two dimensional convolution were presented in Chapter 1, Section 1.9.2. This section will present some practical considerations, routines for convolution, and examples using the routines.

7.3.1 Convolution Speed-Up

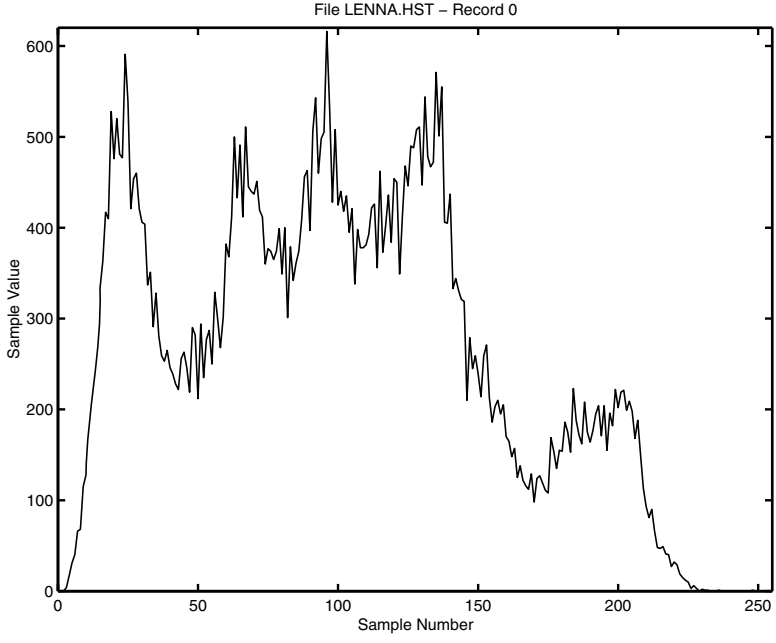
In Chapter 5 it was demonstrated that it is sometimes more efficient to perform one-dimensional convolution by using an FFT and an inverse FFT (see Section 5.9 and program FASTCON). In this section we discuss the time tradeoff for two-dimensional convolution. Convolution can be thought of as a process of sliding a kernel of size M^2 over an image of size N^2. At each translation point the values in the kernel are multiplied by the values that they overlay in the image, and a sum of these products is the resultant matrix value. In most cases, the size of the kernel (M) is less than the size of the image (N), and in many cases the kernel size is much less.

The number of computations required for a two-dimensional convolution (where one multiply/accumulate operation is one computation) is as follows:

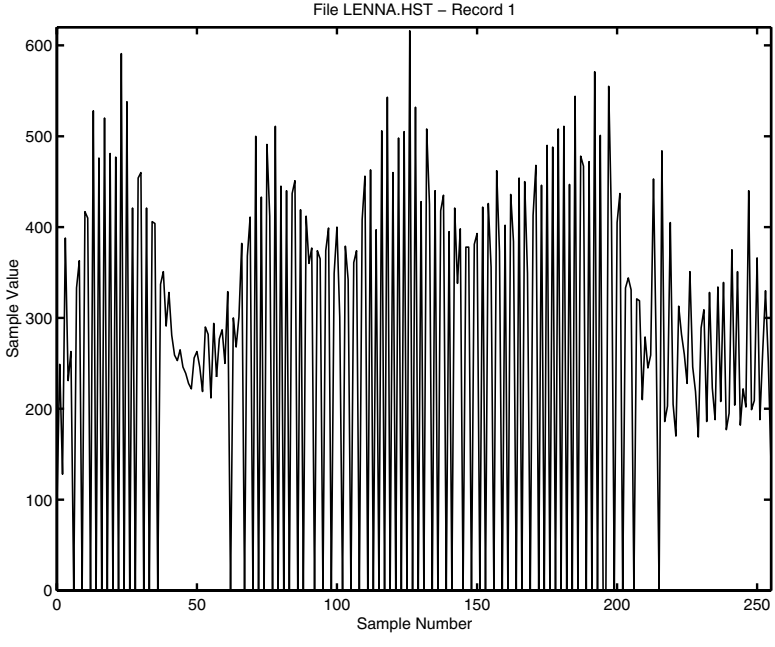
$$\text{2D Convolution Computations} = M^2 (N - M + 1)^2$$

Note that the N^2 term is reduced by the filter size so that only the convolution results obtained with a full convolution sum are counted. When two-dimensional FFT techniques are used the following steps are required:

1. Determine the two-dimensional FFT of the filter kernel by zero padding the filter by the number of zeroes required to make the filter FFT result the same size as the



(a)



(b)

FIGURE 7.11 (a) Histogram of the Lenna test image. (b) Histogram of Lenna image after using the FLATTEN program to make the histogram more uniform.

FIGURE 7.12 Lenna image after histogram flattening using program FLATTEN.

image. The FFT of the filter is stored in memory for repeated use in the convolution.

2. Perform a two-dimensional real FFT on the input image of N x N size. This requires N real FFTs for the rows and $N/2$ complex FFTs for the columns.
3. Multiply the image FFT result (a complex matrix N rows by $N/2$ columns) by the stored filter FFT.
4. Perform a two-dimensional inverse FFT on the complex product to provide a real result (requires the same number of operations as the two-dimensional real FFT).

The total number of real computations (where complex adds are counted as two real computations and complex multiplies are four real computations) required for this process is given by:

$$\text{2D Fast Convolution Computations} = 2\,N^2 + 8\,N^2\log(N)$$

The number of computations for the fast convolution method and the direct method is compared for several image and kernel sizes in Table 7.2. As the table shows, the filter kernel must be larger than 9 x 9 before "fast" techniques are of advantage for a rather small 256 x 256 image size. This result is rather optimistic, since it does not consider the time required to manipulate the data in the FFT algorithm. The actual kernel size where the fast techniques are a benefit is usually 13 x 13 or greater. In many image-processing applications, the filter size required is less than 9 x 9; a 5 x 5 kernel might be considered average. Also, the complexity and cost of real-time hardware required for the FFT computations is usually much greater than the direct convolution hardware. Thus, in most practical image-processing applications, FFT techniques are not used.

TABLE 7.2 Time tradeoff for Fast Convolution in Images. (Entries in tables are number of real computations.)

Kernel Size	Image Size			
	128 x 128		256 x 256	
	Normal	"Fast"	Normal	"Fast"
3 x 3	142,884	950,272	580,644	4,325,376
5 x 5	384,400	950,272	1,587,600	4,325,376
7 x 7	729,316	950,272	3,062,500	4,325,376
9 x 9	1,166,400	950,272*	4,981,824	4,325,376*
11 x 11	1,684,804	950,272	7,322,436	4,325,376

*Indicates where the fast method begins to give a possible advantage.

In Sections 7.3.2 and 7.3.3, the two-dimensional convolution function is discussed and demonstrated using program SMOOTH. Section 7.3.4 illustrates the two-dimensional FFT convolution method using program CON2DFFT.

7.3.2 Two-Dimensional Convolution Function

The two-dimensional convolution function, **convol2d,** convolves a kernel of a given size with an image matrix. The values that must be passed to the function are the pointers to the **Matrix** structures (using C++ references) for the input image matrix and the filter kernel matrix. The function determines the size and type of the matrices from the information in the **Matrix** structures. The function returns a pointer to the resultant **Matrix** structure. Listing 7.9 shows the complete function.

The function should be used with odd-size filter kernels such that both dimensions are odd (but not required to be the same). With an odd length, the output image can be aligned with the input image. In the program, space for the output matrix is first allocated and the dimensions of the kernel are verified. A dead band is determined from the size of the filter kernel. This dead band is the number of rows and columns around the edge of the picture whose pixels must be formed from less than a full kernel-sized compliment of input image pixels. If output values were created for the dead band they would tend to have less than full amplitude values and would exhibit a "washed-out" look sometimes called convolution edge effects. In **convol2d**, the dead band areas are black because the **mOut** matrix is initialized to zero.

Next, a double-nested for loop steps through the input matrix rows and columns. Each point represents an offset position that is used to pull a submatrix of the size of the kernel out of the input matrix. This submatrix is multiplied point by point by the filter kernel matrix. The remaining operation to create the output pixel value is addition of the products. These operations are done by the double for loops, which step through the rows

and columns in the submatrix adding all values to produce a single result. This result is then scaled and placed in the output matrix at a position offset by half the width and height of the kernel.

To illustrate the use of **convol2d,** two examples are used. In the first (Section 7.3.3), **convol2d** is used to apply a Gaussian lowpass filter to an image. This operation is often performed to reduce the high-frequency content in the image for noise removal or as preprocessing before image decimation or edge detection. The advantage of the

```
////////////////////////////////////////////////////////////////////
//
// convol2d( const Matrix<Type>& mIn, const Matrix<Type>& mFilt )
// Filters image with 2D convolution filter.
//
// Throws:
// DSPException
//
////////////////////////////////////////////////////////////////////
template <class Type>
Matrix<Type> convol2d( const Matrix<Type>& mIn, const Matrix<Type>& mFilt )
{
    int row = 0;
    int col = 0;
    Matrix<Type> mOut( mIn.rows(), mIn.cols() );
    mOut = (const Type)0;


    // Set the size of the kernel rows and columns
    int kernel_rows = mFilt.rows();
    int kernel_cols = mFilt.cols();

    int dead_rows = kernel_rows / 2;
    int dead_cols = kernel_cols / 2;

    // Calculate the normalization factor for the kernel matrix
    int normal_factor = 0;
    const Type *fltptr = NULL;
    for( row = 0; row < kernel_rows; row++ )
    {
        fltptr = mFilt[row];
        for( col = 0; col < kernel_cols; col++ )
            normal_factor += abs( fltptr[col] );
    }
```

LISTING 7.9 Function **convol2d** used to convolve a filter kernel with an image matrix (contained in file IMAGE.H). (*Continued*)

```
    // Make sure the normalization factor is not 0
    if( normal_factor == 0 )
        normal_factor = 1;

    for( row = 0; row < mIn.rows() - kernel_rows + 1; row++ )
    {
        Type *outptr = mOut[row + dead_rows];
        for( col = 0; col < mIn.cols() - kernel_cols + 1; col++ )
        {
            int sumval = 0;
            for( int i = 0; i < kernel_rows; i++ )
            {
                const Type *inptr = mIn[i + row] + col;
                fltptr = mFilt[i];
                for( int j = 0; j < kernel_cols; j++ )
                    sumval += *inptr++ * *fltptr++;
            }

            outptr[col + dead_cols] = sumval / normal_factor;
        }
    }
    return( mOut );
}
```

LISTING 7.9 (*Continued*)

Gaussian kernel is that for a given frequency roll off in the spatial frequency domain, a minimum-size kernel can be used in the spatial domain. The size of the kernel affects the time of processing very dramatically (see Table 7.2), and the smaller kernel size is, therefore, an important advantage. In the second example (Section 7.3.5), **convol2d** is used to perform edge detection using a series of 3 x 3 convolutions.

7.3.3 Example Convolution Routine

The program SMOOTH (see Listing 7.10) calls the function **convol2d** and performs a common 3 x 3 Gaussian blur lowpass filter function on an input image in the **DSPFile** matrix format. The program begins by reading the input matrix. The 3 x 3 Gaussian kernel is allocated and defined row by row. The filter coefficient values used are as follows:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{7.9}$$

These values give a good integer approximation of a two-dimensional Gaussian weighting. Next, the output matrix is created by a call to **convol2d.** Finally, the matrix is written to disk. There are several reasons for using a program similar to the SMOOTH program to reduce the high-frequency content in an image. These are summarized below:

```
////////////////////////////////////////////////////////////////
//
// smooth.cpp - Convolve matrix with 3x3 Gaussian filter kernel
// Convolves a matrix with a 3x3 Gaussian filter kernel.
//
// Inputs:
// DSPFile of matrix
//
// Outputs:
// DSPFile of filtered matrix
//
////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "image.h"

int main()
{
    try
    {
        DSPFile dspfile;
        String str;

        // Open the input file
        do getInput( "Enter input file name", str );
        while( str.isEmpty() || dspfile.isFound( str ) == false );
        dspfile.openRead( str );

        // Get input file trailer
        String strTrailer;
        dspfile.getTrailer( strTrailer );

        // Read in matrix and close file
        Matrix<int> mIn;
        dspfile.read( mIn );
        dspfile.close();
```

LISTING 7.10 Program SMOOTH used to convolve an image with a simple 3 x 3 Gaussian blur filter. (*Continued*)

```
        if( mIn.rows() < 3 || mIn.cols() < 3 )
          throw DSPMathException( "Matrix too small for convolution" );

        // Give image information
        cout
            << "Image is " << mIn.rows()
            << " rows by " << mIn.cols() << " columns\n"
            << "Trailer: \n" << strTrailer << endl;

        // Filter matrix
        Matrix<int> mFilt( 3, 3 );
        mFilt[0][0] = 1; mFilt[0][1] = 2; mFilt[0][2] = 1;
        mFilt[1][0] = 2; mFilt[1][1] = 4; mFilt[1][2] = 2;
        mFilt[2][0] = 1; mFilt[2][1] = 2; mFilt[2][2] = 1;

        // Get resultant matrix
        Matrix<int> mOut = convol2d( mIn, mFilt );

        // Get output file
        do getInput( "Enter file to create", str );
        while( str.isEmpty() );
        dspfile.openWrite( str );

        // Convert matrix to PIXELs (0-255 range)
        Matrix<PIXEL> mConv( mOut.rows(), mOut.cols() );
        for( int row = 0; row < mOut.rows(); row++ )
        {
            for( int col = 0; col < mOut.cols(); col++ )
            {
                int val = mOut[row][col];
                if( val > 255 ) mConv[row][col] = 255;
                else if( val < 0 ) mConv[row][col] = 0;
                else mConv[row][col] = val;
            }
        }

        // Write out matrix
        dspfile.write( mConv );
        // Add trailer
        strTrailer += "Convolved with Gaussian 3x3 kernel.\n";
        dspfile.setTrailer( strTrailer );
        dspfile.close();
    }
    catch( DSPException& e )
    {
```

LISTING 7.10 (*Continued*)

```
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 7.10 (*Continued*)

1. Reduce high-frequency noise: If a noise source known to have primarily high-frequency content has corrupted the picture a Gaussian blur can improve signal-to-noise ratio.
2. Reduce interference patterns: If high-frequency interference has corrupted the picture (for instance, a pattern of lines caused by radio frequency interference in television transmission), the frequency of the interference can be determined and the blur kernel sized accordingly.
3. Preprocessing: A common operation when processing images is edge detection. Because this is inherently an ill-posed mathematical problem when applied to raw images (see Marr or Bertero, et al.) it is advantageous to first apply a smoothing function to the image. There is physiological evidence that this sequence of operations is performed in the human eye as part of edge extraction (see Marr).
4. Reduce excessive edge sharpness: Some images have exaggerated edge contrast, and the operation of Gaussian convolution reduces the edge sharpness.

A program for adding noise to an image is used in the remaining examples to demonstrate the operation of the filtering programs. Listing 7.11 shows this program, called ADNOIS2D. The ADNOIS2D program works in a manner similar to the one-dimensional noise program ADDNOISE introduced in Chapter 4, Section 4.7.1. The ADNOIS2D program can generate Gaussian, uniform, or spike noise. The Gaussian and uniform distributions are generated in exactly the same way as in the one-dimensional ADDNOISE program. The spike noise is generated using the Gaussian random number generator but is only added to the image if the random number returned is greater than or less than two standard deviations. This method simulates the "salt and pepper" noise which is often present in poorly transmitted images.

As an illustration of the use of the SMOOTH program, a noisy version of the picture of the baboon (BABOON.DAT) will first be created using the ADNOIS2D program, which adds noise to all the records of a DSP data file. The computer dialogue for running ADNOIS2D to add Gaussian noise with a standard deviation of 32 to BABOON.DAT is as follows:

```
////////////////////////////////////////////////////////////////////////
//
// adnois2d.cpp - Add noise to images
//
////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
////////////////////////////////////////////////////////////////////////
//
// Adds noise to each record of existing image data.
//
////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;
        String strTrailer;

        // Open the input file
        do getInput( "Enter file to add noise to", strName );
        while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        // Get image data - use short int to avoid overflow of PIXEL
        Matrix<short int> matNoise;
        dspfile.read( matNoise );
        dspfile.getTrailer( strTrailer );
        dspfile.close();

        // Get type of noise and noise amplitude
        const int UNIFORM = 0;
        const int GAUSSIAN = 1;
        const int SPIKE = 2;

        int typeNoise = 0;
        getInput(
            "Enter type of noise ( 0 = uniform, 1 = gaussian, 2 = spike )",
            typeNoise,
            UNIFORM,
            SPIKE );

        float multNoise = 0.0f;
        getInput( "Enter noise multiplier", multNoise, 0.0f, 1000.0f );
```

LISTING 7.11 Program ADNOIS2D used to add noise to a two-dimensional image stored in a DSP data file. (*Continued*)

```
        // Add noise to records
        String strType;
        int i = 0;
        int j = 0;
        switch( typeNoise )
        {
        case UNIFORM:
            strType = "Uniform distribution";
            for( i = 0; i < matNoise.rows(); i++ )
            {
                for( j = 0; j < matNoise.cols(); j++ )
                {
                    matNoise[i][j] =
                        round( matNoise[i][j] + ( multNoise * uniform() ) );
                }
            }
            break;
        case GAUSSIAN:
            strType = "Gaussian distribution";
            for( i = 0; i < matNoise.rows(); i++ )
            {
                for( j = 0; j < matNoise.cols(); j++ )
                {
                    matNoise[i][j] =
                        round( matNoise[i][j] + ( multNoise * gaussian() ) );
                }
            }
            break;
        case SPIKE:
            strType = "Spike distribution";
            for( i = 0; i < matNoise.rows(); i++ )
            {
                for( j = 0; j < matNoise.cols(); j++ )
                {
                    double tmpFlt = gaussian();
                    if( tmpFlt > 2.0f )
                        matNoise[i][j] = matNoise[i][j] + multNoise;
                    else if( -tmpFlt > 2.0f )
                        matNoise[i][j] = matNoise[i][j] - multNoise;
                }
            }
            break;
        }

        // Write noisy matrix out to file
        do getInput( "Enter noisy output file name", strName );
```

LISTING 7.11 (*Continued*)

```
        while( strName.isEmpty() );

        dspfile.openWrite( strName );
        dspfile.write( matNoise );

        // Make descriptive trailer and write to file
        char fmtBuf[80];
        sprintf(
            fmtBuf,
            "\n2D noise signal, %s, Amplitude = %f\n",
            strType,
            multNoise );
        // Append to trailer
        strTrailer += fmtBuf;
        dspfile.setTrailer( strTrailer );
        cout << strTrailer << endl;
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }

    return 0;
}
```

LISTING 7.11 (*Continued*)

```
Enter input file name : BABOON.DAT
Enter noise type (0=uniform, 1=gaussian, 2=spike) [0...2] : 1
Enter noise multiplier [0...1e+003] : 32
Enter output matrix file name : BABNSE.DAT
2D noise signal, Gaussian distribution, Amplitude = 32.000000
```

The image of the baboon with noise added is shown in Figure 7.13. After adding noise and creating BABNSE.DAT, the program SMOOTH may be used to try to remove some of the noise in BABNSE.DAT as follows:

```
Enter input file name : BABNSE.DAT
Image is 256 rows by 256 columns
Trailer:
Baboon Image.
2D noise signal, Gaussian distribution, Amplitude = 32.000000

Enter file to create : BABFIL.DAT
```

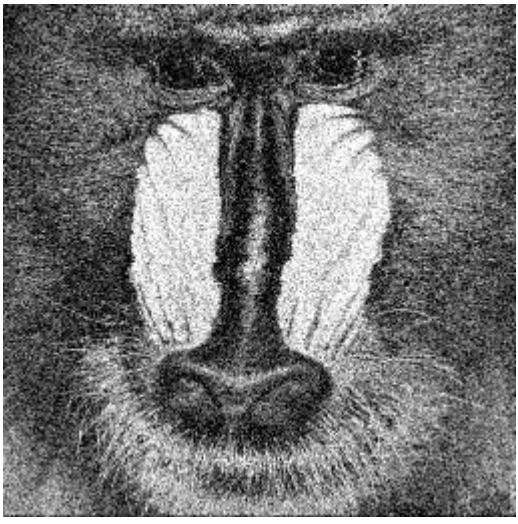


FIGURE 7.13 Baboon image with Gaussian noise ($\sigma = 32$) added.

FIGURE 7.14 Baboon image with noise after application of smoothing filter using program SMOOTH.

The result of applying the Gaussian filter to the noisy image is shown in Figure 7.14. It is clear that some of the higher spatial frequencies (which produce the fine detail in the image) are missing from the filtered image. This is especially evident in the area of the baboon's whiskers. However, when compared to the noisy image, the filtered image clearly has better quality. As in the compression and expansion example, the program MSEOFPIC can be used to get a quantitative measure of the image fidelity. The results are as follows:

```
Image       Root Mean Squared Error Compared to BABOON.DAT
BABNSE.DAT                          32.12
BABFIL.DAT                          21.94
```

The same process was applied to the lenna test image with the results shown in Figure 7.15, the noise added image, and Figure 7.16, the smoothed image. Again, high-frequency edges are sacrificed to improve overall picture fidelity. The fidelity results from MSEOFPIC are

```
Image      Root Mean Squared Error Compared to LENNA.DAT
LENNSE.DAT                          32.12
LENFIL.DAT                          19.82
```

Other levels of Gaussian noise can be added to the BABOON.DAT file using ADNOIS2D, as well as uniform distribution noise and spike noise. *Spike,* or *salt and pep-*

FIGURE 7.15 Lenna image with Gaussian noise ($\sigma = 32$) added.

FIGURE 7.16 Lenna image with noise after application of smoothing filter using program SMOOTH.

per noise, appears in images as black and white specks distributed randomly throughout the image. This can be due to problems in the imaging device or interference from outside signal sources.

Variations can be made to the smoothing filter and the results compared to the 3 x 3 Gaussian kernel. Examples of variations are larger kernel sizes (5 x 5, 7 x 7) and different weighting schemes such as the following

Uniform weighting:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{7.10}$$

Cross shaped kernel:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{7.11}$$

Cross shaped kernel with nonuniform weights:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{7.12}$$

Each of these smoothing kernels have different frequency responses and will, therefore, smooth an image in unique ways. Although mean squared error comparisons are sometimes useful, the evaluation of the performance of an image-smoothing filter is usually done in an qualitative manner by observing the filter results on a set of images.

7.3.4 Two-Dimensional FFT Convolution

The two-dimensional FFT convolution is performed by program CON2DFFT, which is shown in Listing 7.12. The program performs all of the steps in fast 2D convolution outlined in Section 7.3.1. The 2D FFT is performed by applying the one-dimensional FFT to all the rows, transposing the resulting complex matrix and then applying the one-dimensional FFT to all the rows again. Unlike the SMOOTH program, the CON2DFFT program reads the convolution kernel from a **DSPFile**. The kernel can be any size up to the image size and can also be floating point data. Figure 7.17 shows the result of CON2DFFT when a 8 x 8 diamond shaped lowpass filter (file DIAM.DAT) is applied to the LENNA.DAT image file. The 2D filter coefficients in the DIAM.DAT file are as follows:

0	0	0	0	64	0	0	0
0	0	0	64	64	64	0	0
0	0	64	64	64	64	64	0
0	64	64	64	64	64	64	64
0	0	64	64	64	64	64	0
0	0	0	64	64	64	0	0
0	0	0	0	64	0	0	0
0	0	0	0	0	0	0	0

Note that the image in Figure 7.17 is shifted down and to the right by a few pixels and that the 3 pixels along the right and bottom of the original image appear at the left and top of the resulting image. This is because the FFTs used in the fast convolution method wrap around the 3 pixel y-axis and 4 pixel x-axis shift caused by the filter kernel center at location (4,3).

In this example program, complex floating point FFTs are used throughout which slows the 2D convolution operation compared to the integer **convol2d** function. Nevertheless, the CON2DFFT program will typically perform a 19 x 19 floating point convolution on a 256 x 256 image faster than the **convol2d** function can do the same function.

7.3.5 Edge Detection Using Convolution

Edge detection is an important step in most image segmentation techniques. Its purpose is to identify areas of an image where large changes in intensity occur. These changes are often associated with some physical boundary (or edge) in the scene from which the

```
////////////////////////////////////////////////////////////////////////////
//
// con2dfft.cpp - Demonstrate 2D convolution using 2D FFT
//
////////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "dft.h"
#include "disk.h"
#include "get.h"
#include "image.h"

// Maximum FFT size (2^n)
const int MAXFFTSIZE = 10;

////////////////////////////////////////////////////////////////////////////
//
// Does 2D FFT convolution on input image with Gaussian filter
// Gives image data as output.
//
////////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Open the input file
        do getInput( "Enter input signal file", strName );
        while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        // Read from file
        Matrix<float> mSignal;
        dspfile.read( mSignal );
        dspfile.close();

        // Get length of FFT
        int lenIn = mSignal.rows();
        if( mSignal.cols() > lenIn )
            lenIn = mSignal.cols();
        int m = MAXFFTSIZE;
        if( log2( lenIn ) < MAXFFTSIZE )
        {
            getInput(
```

LISTING 7.12 Program CON2DFFT used to demonstrate convolution using 2D FFTs. (*Continued*)

```
            "Enter power of 2 length of FFT",
            m,
            log2( lenIn ),
            MAXFFTSIZE );
    }
    else
    {
        cout
            << "Warning: Truncating signal to "
            << ( 1 << MAXFFTSIZE ) << " samples for FFT\n";
    }

    // Allocate vector large enough to hold data (padded to 2^n)
    int lenFFT = 1 << m;
    Matrix<Complex> mSamp( lenFFT , lenFFT );
    mSamp = (Complex)0.0f;

    Vector<Complex> vSamp( lenFFT );
    Vector<Complex> vSampFFT( lenFFT );
    vSamp = (Complex)0.0f;

    // Find the 2D spectrum of image data
    // Copy data and do FFTs with complex result
    for( int i = 0; i < mSignal.rows(); i++ )
    {
        // Copying from float to Complex
        for( int j = 0; j < mSignal.cols(); j++ )
            vSamp[j].m_real = mSignal[i][j];
        vSampFFT = fft( vSamp, m );
        mSamp.setRow( i, vSampFFT );
    }

    mSamp = transpose( mSamp );

    // Copy data and do complex FFTs with complex result
    for( i = 0; i < lenFFT; i++ )
    {
        vSamp = mSamp.getRow( i );
        vSampFFT = fft( vSamp, m );
        mSamp.setRow( i, vSampFFT );
    }

    // Open the input 2D filter file
    do getInput( "Enter 2D filter file", strName );
    while( strName.isEmpty() || dspfile.isFound( strName ) == false );
    dspfile.openRead( strName );
```

LISTING 7.12 (*Continued*)

```
// Read from file
Matrix<float> mFiltIn;
dspfile.read( mFiltIn );
dspfile.close();

Matrix<Complex> mFilt( lenFFT , lenFFT );
mFilt = (Complex)0.0f;
vSamp = (Complex)0.0f;

// Find the 2D spectrum of the filter
// Copy data and do FFTs with complex result
double Scale = 0.0;
for( i = 0; i < mFiltIn.rows(); i++ )
{
    for( int j = 0; j < mFiltIn.cols(); j++ )
    {
        vSamp[j] = mFiltIn[i][j];
        // add up scale value for final result scale
        Scale += fabs( vSamp[j].m_real );
    }
    vSampFFT = fft( vSamp, m);
    mFilt.setRow( i, vSampFFT );
}

mFilt = transpose( mFilt );

// Copy data and do complex FFTs with complex result
for( i = 0; i < lenFFT; i++)
{
    vSamp = mFilt.getRow( i );
    vSampFFT = fft( vSamp, m);
    mFilt.setRow( i, vSampFFT );
}

// Multiply frequency domain results
mSamp = pwisemult( mSamp, mFilt );

// Find 2D inverse FFT of 2D spectrum
// Copy data and do inverse FFTs with complex result
for( i = 0; i < lenFFT; i++ )
{
    vSamp = mSamp.getRow( i );
    vSampFFT = ifft( vSamp, m );
    mSamp.setRow( i, vSampFFT );
}

mSamp = transpose( mSamp );
```

LISTING 7.12 (*Continued*)

```
        // Copy data and do inverse FFTs with complex result
        for( i = 0; i < lenFFT; i++ )
        {
            vSamp = mSamp.getRow( i );
            vSampFFT = ifft( vSamp, m );
            mSamp.setRow( i, vSampFFT );
        }

        // round real part of complex result and clip
        Matrix<PIXEL> mOut( lenFFT , lenFFT );
        float InvScale = 1.0f / Scale;
        for( i = 0; i < lenFFT; i++ )
        {
            for( int j = 0; j < lenFFT; j++ )
            {
                float x = InvScale * mSamp[i][j].m_real;
                if( x < 0.0f ) x = 0.0f;
                else if( x > 255.0f ) x = 255.0f;
                mOut[i][j] = (PIXEL)round(x);
            }
        }

        // Write result image to file
        do getInput( "Enter PIXEL image from iverse FFT to create", strName );
        while( strName.isEmpty() );
        dspfile.openWrite( strName );
        dspfile.write( mOut );

        // Make descriptive trailer for output image file
        String strTrailer;
        getInput( "Enter trailer string", strTrailer );

        if( strTrailer.isEmpty() == false )
            dspfile.setTrailer( strTrailer );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 7.12 (*Continued*)

FIGURE 7.17 Image of Lenna after application of 8 x 8 DIAM.DAT 2D filter using program CON2DFFT.

image is derived. The ultimate object in segmentation is to develop a contiguous set of pixels forming a closed contour around each object of interest in the image. Edge detection alone is not sufficient to this task but must usually be combined with more heuristic rules-based algorithms which track boundaries and link edges to form whole contours (see Pratt).

There is physiological evidence that edge detection is an important step in the early vision process for humans (see Marr). The images formed by edge detection preceded by Gaussian blur filtering of various sigma values (widths of the Gaussian lobe) have been used in successful segmentation algorithms even in quite complex images.

One approach to edge detection is shown in Figure 7.18. Here the original image is passed through a Gaussian filter which gives an output with a controlled frequency content. As the size of sigma in the Gaussian kernel is increased, the image is increasingly blurred and the edges are actually de-emphasized. However, if this first step is skipped, the edge detected image will often have numerous extraneous detection points which overly complicate the edge linking algorithms which follow and may make further processing impractical. Following the Gaussian filter is the edge enhancement process. There are several possible algorithms used here which will be discussed below. The final step is the actual detection thresholding in which the edge enhanced image is compared point by point to a threshold value. This step produces a two-valued image where one pixel value represents the presence of an edge, and the other represents absence of an edge.

The algorithms used in the edge enhancement step generally involve taking the difference of pixels in the region of interest. There is evidence that the human eye has edge-detection structures that perform a function similar to the Laplacian (see Marr). In discrete systems, the Laplacian is composed of the second differences of the sampled signal.

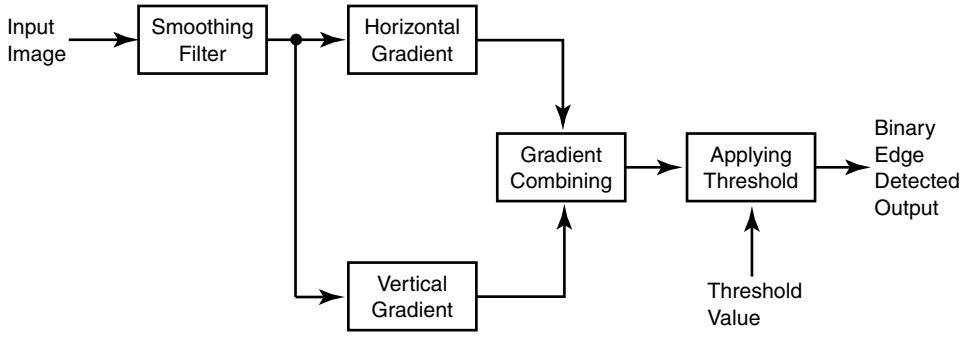


FIGURE 7.18 Operations in edge detection.

There are several approximations to the Laplacian in two dimensions. One commonly used kernel is

$$\begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix} \tag{7.13}$$

Another approximation that requires less computation (due to the 0 values in the kernel) is

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{7.14}$$

Note that the two kernels have different gain values, and the results would have to be scaled for comparison. Both of the kernels shown above will emphasize edges in the picture to the total exclusion of areas of constant intensity. A 3 x 3 region of the picture whose pixel values are all equal will produce a zero output from these kernels. In applications where less drastic edge enhancement is desired, the negative value at the kernel center is reduced in magnitude. One technique is to subtract the result of the Laplacian convolution from the original image values. This produces a picture with much of the original information but having increased edge contrast.

The arguments against using the Laplacian are its tendency to produce extraneous edge points and give especially strong response to corners. The results of a Laplacian edge enhancement are often noisy. Because of these problems, combinations of gradient kernels are often used instead. The discrete gradient is simply the first difference of the image pixel values at locations in the picture. In order to create output images that can register with the input image, the central differences are most often used. A kernel that produces central first differences is as follows:

$$\begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \tag{7.15}$$

This kernel has a large response for vertical edges but no response to horizontal edges. For example, trying this kernel on a section of an image with the following values:

$$\begin{bmatrix} 128 & 128 & 128 & 10 & 10 & 10 \\ 128 & 128 & 128 & 10 & 10 & 10 \\ 128 & 128 & 128 & 10 & 10 & 10 \\ 128 & 128 & 128 & 10 & 10 & 10 \end{bmatrix} \tag{7.16}$$

Its response will be

$$\begin{bmatrix} 0 & 0 & 118 & 118 & 0 & 0 \\ 0 & 0 & 118 & 118 & 0 & 0 \\ 0 & 0 & 118 & 118 & 0 & 0 \\ 0 & 0 & 118 & 118 & 0 & 0 \end{bmatrix} \tag{7.17}$$

which shows a high-amplitude response to the vertical edge. The corresponding kernel for horizontal edges is

$$\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \tag{7.18}$$

which will produce no response for vertical edges. The drawback of both these kernels is that they also respond just as strongly to isolated points as to edges. To reduce this problem the kernels are usually extended to 3 x 3 or greater size in the following manner:

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \tag{7.19}$$

for vertical edges and

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{7.20}$$

for horizontal edges. These two kernels reduce the response to isolated points but tend to blur the picture, since they are area operators. A compromise between the simple first difference operators and the 3 x 3 operators above is the Sobel edge detection operator whose kernel is as follows:

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{7.21}$$

The transposed version is used for the horizontal operator. The Sobel operators have the advantage of emphasizing the central part of the edge while still reacting less strongly than the 1 x 3 operator to isolated points.

If each direction of the Sobel operator is applied to an image, two new images are created: one showing the vertical response and the other showing the horizontal response. Since our purpose is to determine the existence and location of edges in a picture we would like to combine these two images into a single one. The best way of doing this must take into account the combined response of the operators to diagonal edges as well as vertical and horizontal.

A diagonal edge will show equal response to the Sobel operators in both orientations. For example, the following diagonal edge:

$$\begin{bmatrix} 20 & 10 & 10 & 10 \\ 20 & 20 & 10 & 10 \\ 20 & 20 & 20 & 10 \\ 20 & 20 & 20 & 20 \end{bmatrix} \tag{7.22}$$

if extended in both directions, will give the following response to the vertical Sobel:

$$\begin{bmatrix} 30 & 30 & 10 & 0 & 0 & 0 \\ 10 & 30 & 30 & 10 & 0 & 0 \\ 0 & 10 & 30 & 30 & 10 & 0 \\ 0 & 10 & 30 & 30 & 10 & 10 \\ 0 & 0 & 10 & 30 & 30 & 10 \\ 0 & 0 & 0 & 10 & 30 & 30 \end{bmatrix} \tag{7.23}$$

and a similar amplitude response to the horizontal Sobel. We would like the maximum of the combined response to be equal to the maximum of the combined response of an identical amplitude vertical or horizontal edge (which would be 40). The following three methods might be used

1. Compute the square root of the sum of the squares of the horizontal and vertical responses. This views the responses as quadrature components of a vector and combines them accordingly.
2. Take the sum of the absolute values of the components. This is a simpler computation and probably not as accurate.
3. Take the maximum of the magnitudes of the two components. From the above example this is close to the proper value.

These three methods of edge response combining are compared for the Sobel response to vertical, horizontal, and diagonal edges in Table 7.3. Although the square root of the sum method gives more equal response across the three-edge orientations, the maximum of the absolute values is often chosen because of its simpler computation and, therefore, faster execution.

7.3.6 Edge Detection Routine

The program for edge detection using convolution is called EDETECT and shown in Listing 7.13. The following steps are used for edge detection in this program.

1. Convolution with a 3 x 3 Gaussian blur kernel.
2. Edge finding with a 3 x 3 vertical Sobel.
3. Edge finding with a 3 x 3 horizontal Sobel.
4. Combining the two Sobel operator outputs using the largest absolute value technique described above.
5. Detection of the combined matrix using a user specified threshold.

The EDETECT program begins by reading a matrix from disk. A filter kernel matrix is then allocated, and the coefficients are set to Gaussian kernel values. The convolution is performed with the result stored in a **Matrix** structure. Next, two output matrices are formed by convolution of the blurred image with first a vertical and then a horizontal Sobel kernel.

The Sobel operator outputs are combined by comparing their absolute values at each point and selecting the larger of the two for placement in the combined matrix. Note that the direction of the change in intensity is lost in this operation, since the sign information of the Sobel is removed. Because this program is intended to produce only segmentation based on existence of edges, no necessary information is sacrificed in this operation.

The user is next asked to select a threshold for the edge strength. Each pixel in the combined gradient output is then compared to the user-specified threshold, and either a black or white pixel is placed according to the result of the comparison. A white (255) on

TABLE 7.3 Comparison of Methods of Combining Vertical and Horizontal Sobel Edge Detector Responses. (For an edge of amplitude *A* the combining methods for the Sobel operator have the following responses to different edge orientations.)

	Edge Orientation		
	Horizontal	**Vertical**	**Diagonal**
Root mean square	*A*	*A*	1.06 x *A*
Sum of absolutes	*A*	*A*	2.00 x *A*
Max of absolutes	*A*	*A*	0.75 x *A*

```
////////////////////////////////////////////////////////////////////////
//
// edetect.cpp - Detect edges in images
//
////////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "image.h"
////////////////////////////////////////////////////////////////////////
//
// Performs edge detection on an image in the DSPFile
// format where each row in the image is a record of the file.
//
// The steps in the processing are:
//     1. Convolve with a Gaussian blur kernel.
//     2. Find edges with a vertical Sobel operator.
//     3. Find edges with a horizontal Sobel operator.
//     4. Choose highest magnitude of the Sobel operators.
//     5. Detect edges using a threshold value.
//
////////////////////////////////////////////////////////////////////////
int main()
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Open the input file
        do getInput( "Enter image file to be edge detected", strName );
        while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        Matrix<int> mIn;
        dspfile.read( mIn );
        dspfile.close();

        int size = mIn.rows() * mIn.cols();

        // Create convolution matrix
        Matrix<int> mFilt( 3, 3 );

        // Set the values of the filter matrix to a Gaussian kernel
        mFilt[0][0] = 1; mFilt[0][1] = 2; mFilt[0][2] = 1;
```

LISTING 7.13 Program EDETECT used for edge detection of an image using the **convol2d** function. (*Continued*)

```
mFilt[1][0] = 2; mFilt[1][1] = 4; mFilt[1][2] = 2;
mFilt[2][0] = 1; mFilt[2][1] = 2; mFilt[2][2] = 1;

// Perform the Gaussian convolution
Matrix<int> mOut = convol2d( mIn, mFilt );

// Vertical Sobel Operator
mFilt[0][0] = 1; mFilt[0][1] = 0; mFilt[0][2] = -1;
mFilt[1][0] = 2; mFilt[1][1] = 0; mFilt[1][2] = -2;
mFilt[2][0] = 1; mFilt[2][1] = 0; mFilt[2][2] = -1;

// Convolve the smoothed matrix with the vertical Sobel kernel
Matrix<int> mVS = convol2d( mOut, mFilt );

//Horizontal Sobel Operator
mFilt[0][0] = 1; mFilt[0][1] = 2; mFilt[0][2] = 1;
mFilt[1][0] = 0; mFilt[1][1] = 0; mFilt[1][2] = 0;
mFilt[2][0] = -1; mFilt[2][1] = -2; mFilt[2][2] = -1;

// Convolve the smoothed matrix with the horizontal Sobel kernel
Matrix<int> mHS = convol2d( mOut, mFilt );

// Take the larger of the magnitudes of the two matrices,
// over write the OUT matrix with the result, while finding
// the min, max and average gradient values
int min = 10000;
int max = 0;
float avg = 0.0f;
for( int i = 0; i < mHS.rows(); i++ )
{
    for( int j = 0; j < mHS.cols(); j++ )
    {
        int temp1 = (int)abs( mHS[i][j] );
        int temp2 = (int)abs( mVS[i][j] );
        if( temp1 < temp2 ) temp1 = temp2;
        if( temp1 < min ) min = temp1;
        if( temp1 > max ) max = temp1;
        avg += temp1;
        mOut[i][j] = temp1;
    }
}
avg /= size;

// Compare the magnitudes of the edges to a threshold
```

LISTING 7.13 (*Continued*)

```
        cout << "The min, max and average of the gradients are:\n";
        cout << " Min = " << min << endl;
        cout << " Max = " << max << endl;
        cout << " Avg = " << avg << endl;

        // Get the threshold value as a user input
        int threshold = 0;
       getInput( "Enter value of detection threshold", threshold, min, max );

        // Compare each edge value in the edge matrix with the user supplied
        // threshold and create detection matrix (white on black)
        Matrix<PIXEL> mDetect( mOut.rows(), mOut.cols() );
        for( i = 0; i < mOut.rows(); i++ )
        {
            for( int j = 0; j < mOut.cols(); j++ )
            {
                if( mOut[i][j] > threshold ) mDetect[i][j] = 255;
                else mDetect[i][j] = 0;
            }
        }

        // Write matrix out to file
        do getInput( "Enter edge image file name", strName );
        while( strName.isEmpty() );

        dspfile.openWrite( strName );
        dspfile.write( mDetect );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
    return 0;
}
```

LISTING 7.13 (*Continued*)

black (0) type display of the edges is generated. The final operation is to write the matrix of edge-detection boundaries out to disk.

To illustrate the process of edge detection the program EDETECT is run on the BABOON.DAT test image as shown in the following computer dialogue:

```
Enter image file to be edge detected : BABOON.DAT
The min, max and average of the gradients are:
    Min =    0
    Max =    91
    Ave =    9.2
Enter value of detection threshold [0...91] : 20
Enter detection image file name : BABDET.DAT
```

The resulting edge-detected image is shown in Figure 7.19. The strengths and weaknesses of edge-detection routines are illustrated in this picture. The clear outlines in the original picture are well defined in the edge-detected version. The outline of the nose and cheeks is clearly traced. However, numerous other unconnected points also appear in the image, for instance, those associated with the whiskers. Because of this quality in edge-detected images, it is necessary to perform further steps for complete segmentation of images. Following are some of the techniques used.

1. Segment joining programs that search for nearby edge segments with similar orientations. When two segments are found that meet the proximity and orientation criteria, they are joined into a single, longer segment.

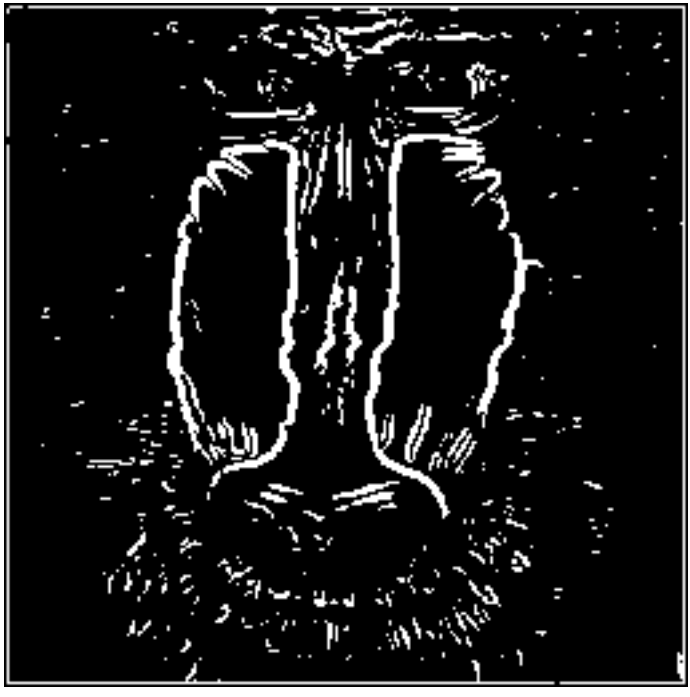


FIGURE 7.19 Edge detected baboon image generated using program EDETECT.

FIGURE 7.20 Edge-detected Lenna image generated using program EDETECT.

2. Edge-tracking programs that search for a maximum gradient within certain forward-directed region of pixels and continually extend the current edge by repeating this search.
3. Region dilation programs that cause edge segments to touch by growing them on successive min or max finding passes (see Section 7.4).

An image more easily segmented by the EDETECT program is the Lenna test image. Because less of the image area is devoted to fine detail and more to large, similar intensity areas, Figure 7.20 shows a more complete segmentation of the Lenna image than achieved with baboon image.

7.4 NONLINEAR PROCESSING OF IMAGES

This section presents a function for filtering by sorting as discussed in Chapter 1, Section 1.5. The advantages and techniques for median filtering were discussed in more detail in Section 1.5.2 and will not be repeated here. The function **nonlin2d** can be used for median, min, or max filtering. A complete listing of the function is given in Listing 7.14.

The **nonlin2d** function accepts a pointer to the matrix structure to be filtered, the size of the kernel to use for filtering, and the type of filter (min, max, median) to be per-

```
////////////////////////////////////////////////////////////////////
//
// nonlin2d( const Matrix<PIXEL>& mIn, int type, int kernelSize )
// Filters image with non-linear filter ( erosion, dilation,
// and median ) on an image matrix of data using a square kernel.
//
// Note:
// The filter types can be one of:
//     NONLINEROSION
//     NONLINDILATION
//     NONLINMEDIAN
//
// The kernel size must be odd.
//
// Throws:
// DSPException
//
// Returns:
// Matrix<PIXEL> of filtered image.
//
////////////////////////////////////////////////////////////////////
Matrix<PIXEL> nonlin2d( const Matrix<PIXEL>& mIn, int type, int kernelSize )
{
    // Unknown filter type
    if( type < NONLINEROSION || type > NONLINMEDIAN )
        throw DSPParamException( "Unknown filter type" );

    // Kernel size be greater than one and odd
    if( kernelSize <= 1 || ( kernelSize % 2 ) == 0 )
        throw DSPParamException( "Kernel size incorrect" );

    // Create return matrix
    Matrix<PIXEL> mOut( mIn.rows(), mIn.cols() );
    mOut = (PIXEL)0;

    // Set the dead space between kernel and edge
    int deadSize = kernelSize / 2;

    // Allocate memory for the sorting array
    int kernelLen = kernelSize * kernelSize;
    PIXEL *filtArray = new PIXEL[kernelLen];
    if( filtArray == NULL )
        throw DSPMemoryException();
```

LISTING 7.14 Function **nonlin2d** used to perform max (dilation), min (erosion), or median nonlinear filtering of an image matrix (contained in file IMAGE.CPP). (*Continued*)

```
    for( int i = 0; i < kernelLen; i++ )
        filtArray[i] = (PIXEL)0;

    // Perform the sort on each submatrix in the input matrix
    for( int row = 0; row < mIn.rows() - kernelSize + 1; row++ )
    {
        for( int col = 0; col < mIn.cols() - kernelSize + 1; col++ )
        {
            // Get block of values from original matrix
            PIXEL *arrayptr = filtArray;
            for( int subRow = row; subRow < row + kernelSize; subRow++ )
            {
              for( int subCol = col; subCol < col + kernelSize; subCol++ )
                    *arrayptr++ = mIn[subRow][subCol];
            }
            qsort( filtArray, kernelLen, sizeof( PIXEL ), pixelcmp );
            switch( type )
            {
            case NONLINEROSION:
                mOut[row + deadSize][col + deadSize] = filtArray[0];
                break;
            case NONLINDILATION:
                mOut[row + deadSize][col + deadSize] = filtArray[kernelLen - 1];
                break;
            case NONLINMEDIAN:
                mOut[row + deadSize][col + deadSize] = filtArray[kernelLen / 2];
                break;
            }
        }
    }
    delete [] filtArray;
    return( mOut );
}
////////////////////////////////////////////////////////////////////
//
// Comparison functions
//
////////////////////////////////////////////////////////////////////
int pixelcmp( const void *a, const void *b )
{
    if( a == NULL || b == NULL )
        return 0;
    return( *(PIXEL *)a - *(PIXEL *)b );
}
```

LISTING 7.14 (*Continued*)

formed. The return value of the function is a pointer to the filtered output matrix. The function **nonlin2d** is very similar to **convol2d** (see Section 7.2). The size of the dead band due to edge effects is determined from the kernel size and a submatrix is copied into a linear array which allows the use of the standard library function **qsort.** The **pixelcmp** function specified for use with **qsort** causes the values to be arranged from smallest to largest. A **switch** statement is then used to select the value from the sort depending on the type of filter to be performed.

The NONLIN2D program illustrates the use of the **nonlin2d** function. This program provides erosion and dilation of image areas or median filtering of images using the **nonlin2d** function min, max, and median capabilities. Erosion and dilation are methods of region growing or shrinking in image processing. These techniques are used in segmentation when the relative gray level of the areas of interest are known with respect to the background. If, for instance, dark blobs are present in an image and appear against a lighter background, the size of the dark areas would be increased by successive applications of a minimum picking filter. Conversely, if light areas are of interest they can be increased in size with a maximum picking filter. In addition to the increase in size of the areas against the background, the consistency of the darkness or lightness is increased. The dark blobs will tend to have more homogeneous gray levels after application of a minimum filter.

Median filtering is often used in image processing to remove noise from an image. It is preferred to a blur kernel because of its edge retaining properties and is very good at removing "salt and pepper" or impulse spike noise. The program NONLIN2D is shown in Listing 7.15. The first step is to read the input file and create a **Matrix** structure for the

```
////////////////////////////////////////////////////////////////////
//
// nonlin2d.cpp - Non-linear filtering of images
// Program to filter images using min, max,
// or median filters.
//
// Inputs:
// DSPFile to image filter
//
// Outputs:
// Filtered image DSPFile
//
////////////////////////////////////////////////////////////////////
#include "dsp.h"
#include "disk.h"
#include "get.h"
#include "image.h"

int main()
```

LISTING 7.15 Program NONLIN2D used to perform nonlinear filtering of an image file. (*Continued*)

```
{
    try
    {
        DSPFile dspfile;
        String strName;

        // Get file name
        do getInput( "Enter input file name", strName );
        while( strName.isEmpty() || dspfile.isFound( strName ) == false );
        dspfile.openRead( strName );

        Matrix<PIXEL> mIn;
        dspfile.read( mIn );
        dspfile.close();

        // Get filter type
        int ft, passes;
        cout << "Enter the function desired\n";
        cout << " 1 – Erosion of light areas (min)\n";
        cout << " 2 – Dilation of light areas (max)\n";
        cout << " 3 – Median filter of the image\n";
        getInput( "Enter filter choice", ft, NONLINEROSION, NONLINMEDIAN );

        // Get number of passes
        getInput( "Enter number of passes", passes, 1, 10 );

        // Filter image for each pass
        Matrix<PIXEL> mOut = mIn;
        // default size of filter kernel is 3
        for( int i = 0; i < passes; i++ )
            mOut = nonlin2d( mOut, ft );

        do getInput( "Enter output file name", strName );
        while( strName.isEmpty() );
        dspfile.openWrite( strName );
        dspfile.write( mOut );
        dspfile.close();
    }
    catch( DSPException& e )
    {
        // Display exception
        cerr << e;
        return 1;
    }
 return 0;
}
```

LISTING 7.15 (*Continued*)

image values. The user is then given a choice between erosion and dilation of the light areas or median filtering of the image. The user is also given the opportunity to specify multiple passes of whichever filter is chosen. The filter size is set to three and the filter type in the call to **nonlin2d** is chosen based on user input. The selected number of passes is performed by the loop.

7.4.1 Median Filtering

This example illustrates the use of the median filter option of NONLIN2D on an image corrupted with spike noise. To create the corrupted image the program ADNOIS2D is run as follows:

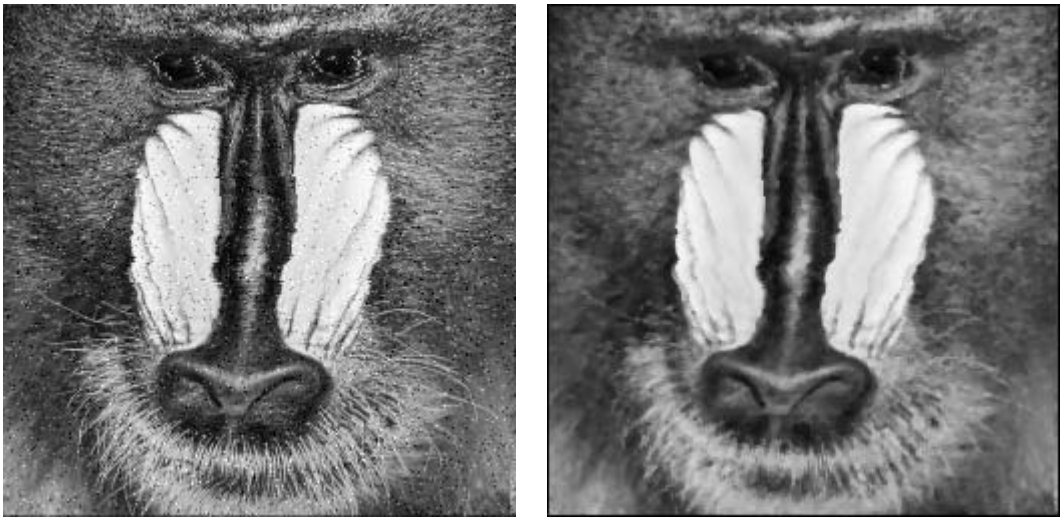
```
Enter input file name : BABOON.DAT
Enter noise type (0=uniform, 1=gaussian, 2=spike) [0...2] : 2
Enter noise multiplier [0...1e+003] : 64
Enter output matrix file name : BABSPK.DAT
2D noise signal, Spike distribution, Amplitude = 64.000000
```

This produces an image of black and white specks shown in Figure 7.21(a). To apply median filtering to this image the NONLIN2D program is run with the following responses at the prompts:

```
Enter name of input image file : BABSPK.DAT
Enter the function desired:
    1 – Erosion of light areas (min).
    2 – Dilation of light areas (max).
    3 – Median filter of the image.
Enter your choice [1...3] : 3
Enter number of passes [1...10] : 1
Enter name of filtered output file : BABMED.DAT
```

Figure 7.21(b) shows the median filtered image of the noisy baboon image (BABMED.DAT). There is some loss of edge information, although this tends to be less severe than that produced by the program SMOOTH, which applies a 3 x 3 Gaussian-shaped convolution kernel. However, the spikes are removed much more effectively than would be possible by linear convolution. This is due to the unusual statistics of salt and pepper noise, which show large errors widely dispersed in the image. A good example of the effectiveness of the median filter in this circumstance can be made by running the SMOOTH program (see Section 7.3.3) on BABSPK.DAT. When this is done, the resulting image has slightly more edge loss than the median filtered BABOON but not nearly as much reduction in the spike noise.

The median process described above is repeated for the Lenna image and the results are shown in Figures 7.22(a) and 7.22(b). The large constant intensity areas in the Lenna image make the noise removal effect even more dramatic.



(a) (b)

FIGURE 7.21 (a) Baboon image with spike noise added (64 amplitude) using program ADNOIS2D. (b) Baboon image with noise after application of median filter using program NONLIN2D.

(a) (b)

FIGURE 7.22 (a) Lenna image with spike noise added (64 amplitude) using program ADNOIS2D. (b) Lenna image with noise after application of median filter using program NONLIN2D.

7.4.2 Erosion and Dilation

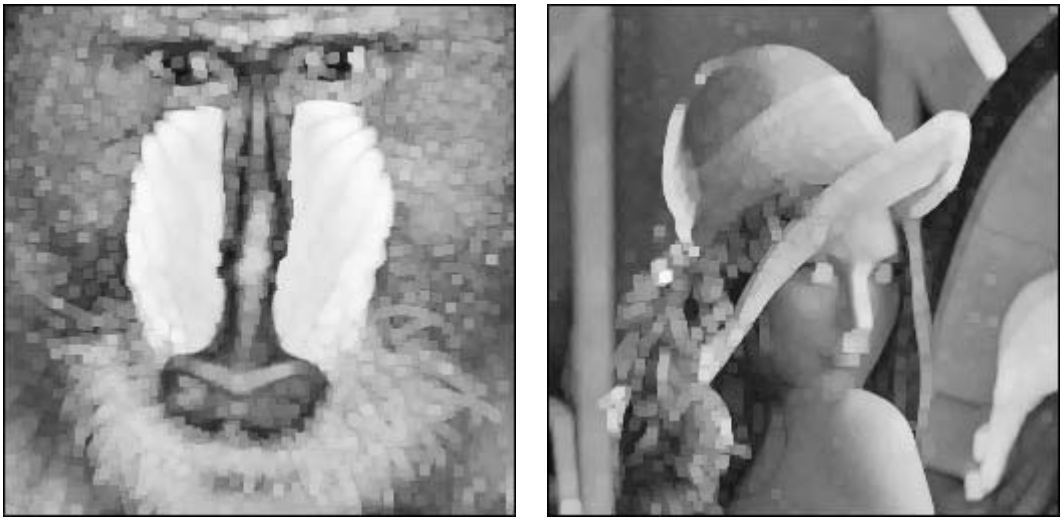
The NONLIN2D program can also be used to find the minimum or maximum value in the 3 x 3 window. The minimum finding filter tends to make the dark areas of an image larger and the bright areas smaller. This type of minimum filter is called an *erosion filter.* On the other hand, the maximum filter is called a *dilation filter* because it makes the bright areas larger and the dark areas smaller. By repeating the min or max operations, the erosion or dilation effects becomes much more pronounced. The following example computer dialogue shows the use of the NONLIN2D program for two passes of the minimum (erosion) filter on the baboon image:

```
Enter name of input image file : BABOON.DAT
Enter the function desired:
        1 – Erosion of light areas (min).
        2 – Dilation of light areas (max).
        3 – Median filter of the image.
Enter your choice [1...3] : 1
Enter number of passes [1...10] : 2
Enter name of filtered output file : BABERD.DAT
```

The resulting baboon image after the above erosion process is shown in Figure 7.23a. The result of the same erosion on the lenna example image is shown in Figure 7.23b. Figures

(a) (b)

FIGURE 7.23 (a) Baboon image after two passes of minimum filtering (erosion) using the program NONLIN2D. (b) Lenna image after two passes of minimum filtering (erosion) using the program NONLIN2D.



(a) (b)

FIGURE 7.24 (a) Baboon image after two passes of maximum filtering (dilation) using the program NONLIN2D. (b) Lenna image after two passes of maximum filtering (dilation) using program NONLIN2D.

7.24a and 7.24b show the result of the maximum dilation filter for the baboon and lenna images. Note that although the images are greatly distorted, the general shape of the large white and dark areas is preserved. Because of this property, erosion and dilation filters are sometimes used in image segmentation and feature extraction.

For example, in image segmentation, dilation followed by erosion can be used to enhance the contrast of sections of an image to the point where a simple threshold can be used to separate one nonoverlapping object from another. If coupled with edge detection, isolated objects can be located precisely. This technique is sometimes used in rapid industrial inspection of parts on an assembly line.

In feature extraction, the noise in an image often can make the estimation of an image feature difficult. In some cases, a more accurate estimate of the overall area of an object can be determined by counting the number of pixels above a threshold after dilation (assuming the object is brighter than the background). A measure of the amount of dirt on an object can be found by counting the number of dark pixels in an image after erosion.

7.5 EXERCISES

1. Use MAKMAT (contained on the enclosed disk) to create a matrix for input to the image processing programs (type MAKMAT < PT.IN to create your own custom matrix by editing PT.IN). Run the DCTTEST program using the test matrices developed with MAKMAT as

inputs or use the provided files as shown below. The resulting transformed matrix is displayed. Write the output to a file (for example DCT.OUT). How do the DCT frequency results change? Which coefficients are small and can be eliminated to compress the data? The sample matrices (provided as files on the disk) can also be used as input to the DCTTEST program. They are as follows:

```
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0
 0  0 64 64 64 64  0  0
 0  0 64 64 64 64  0  0          SQR.DAT
 0  0 64 64 64 64  0  0
 0  0 64 64 64 64  0  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0

 0  0  0  0  0  0  0 64
 0  0  0  0  0  0 64 64
 0  0  0  0 0  64 64 64
 0  0  0  0 64 64 64 64          DIAG.DAT
 0  0  0 64 64 64 64 64
 0  0 64 64 64 64 64 64
 0 64 64 64 64 64 64 64
64 64 64 64 64 64 64 64

 0  0  0  0 64  0  0  0
 0  0  0 64 64 64  0  0
 0  0 64 64 64 64 64  0
 0 64 64 64 64 64 64 64          DIAM.DAT
 0  0 64 64 64 64 64  0
 0  0  0 64 64 64  0  0
 0  0  0  0 64  0  0  0
 0  0  0  0 0   0  0  0
```

2. Run the IDCTTEST program using the transformed matrix from DCTTEST as inputs (e.g., DCT.OUT). The resulting recovered matrix is displayed (it can also be written to a file for further transformations). Compare the recovered matrix with the original matrices. Where do the errors come from? Create larger and more complex matrices to use as inputs and repeat the above steps.
3. Modify the COMPRESS program such that additional DCT coefficients are eliminated. This can be done by changing some of the vOut[pos++] = … statements to vOut[pos++] = 0. How does this change the appearance of the image after running the EXPAND program? How much additional compression can be achieved and still maintain a reasonable image quality?
4. Run the SMOOTH program on each of the test matrices creating in Exercises 1 and 2. Generate a separate output file for each. Use MATPRINT to display the results of the SMOOTH program.
5. Modify SMOOTH to add more kernels, or modify the existing ones and repeat Exercise 3 on these new kernels. For example, try the filter kernels given in Sections 7.3.3 and 7.3.4. It is

also possible to modify SMOOTH to read an input file to provide a different convolution kernel. How do the results vary with the different 2D filters?

6. Using the modified SMOOTH program used in Exercise 4, filter the LENNA.DAT and BABOON.DAT images using the new filter kernels. For example, try the filter kernels given in Sections 7.3.3 and 7.3.4. How do the results vary with the different 2D filters? Which filters are highpass filters and which are lowpass filters?
7. Modify the EDETECT program to use the root mean square method of combining the vertical and horizontal Sobel outputs. Is the edge detection result with the LENNA.DAT image better with this method? Add noise to the lenna image and compare the methods with noise added.
8. Using the NONLIN2D program, compare the ability of median filtering to remove noise from an image to the filtering obtained using the SMOOTH program used in Exercise 4. How do the results vary with the different 2D filters?

7.6 REFERENCES

BERTERO, M., POGGIO, T. A., and TORRE, V., "Ill-Posed Problems in Early Vision," *Proceedings of the IEEE,* August 1988

CHEN, W., SMITH, C. H., and FRALICK, S., "A Fast Computational Algorithm for the Discrete Cosine Transform," *IEEE Trans. Commun.,* COM-25, 1977.

GONZALEZ, RAFAEL C., and WINTZ, P., *Digital Image Processing,* Addison-Wesley, Reading, MA, 1977.

HORN, B., and KLAUS, P., *Robot Vision,* MIT Press, 1986.

MARR, D., *Vision,* W.H. Freeman., 1982.

MITCHELL, O. R., and GROGEN, T. A., "Evaluation of Fourier Descriptors for Target Recognition in Digital Imagery," *Final Technical Report, RADC-TR-83-33,* Rome Air Development Center, February 1983.

PERSOON, E., and FU, K. S., "Shape Discrimination Using Fourier Descriptors," *IEEE Trans. Syst., Man, Cybern.* vol. SMC-4, July 1974.

PRATT, WILLIAM K., *Digital Image Processing,* Wiley-Interscience, New York, 1978.

ROSENFELD, A., "Computer Vision: Basic Principles," *Proceedings of the IEEE,* August 1988.

ROSENFELD, A., and KAK, A. C., *Digital Picture Processing,* Vols. I and II, Academic Press, New York, 1982.

WAHL, F. M., *Digital Image Signal Processing,* Artech House, 1987.

Standard C++ Class Library

This appendix describes the subset of the standard C++ class library which is used in the book or useful in DSP applications. All of the programs contained in this book and the enclosed disk were written using the Microsoft Visual C++ definition of the standard function library described here.

A.1 MATH FUNCTIONS

Most of the following math functions return double values and have double arguments. They are declared (with function prototypes) in the MATH.H include file. This include file should be included (with the directive **#include <math.h>**) at the beginning of any program which uses the math functions. The functions **atof**, **atol**, and **atoi** are declared in the header file STDLIB.H along with several of the other frequently used functions.

A.1.1 Trigonometric Functions: sin, cos, tan, asin, acos, atan, atan2

```
double sin(double x)

double cos(double x)

double tan(double x)

double asin(double x)

double acos(double x)
```

```
double atan(double x)

double atan2(double x, double y)
```

sin, cos and **tan** return trigonometric functions of radian arguments. **asin** returns the arc sin in the range $-\pi/2$ to $\pi/2$. **acos** returns the arc cosine in the range 0 to π. **atan** returns the arc tangent of x in the range $-\pi/2$ to $\pi/2$. **atan2** returns the arc tangent of x/y in the range $-\pi$ to π.

A.1.2 Exponential, Log, Power, Square Root: `exp, log, log10, pow, sqrt`

```
double exp(double x)

double log(double x)

double log10(double x)

double pow(double x, double y)

double sqrt(double x)
```

exp returns the exponential function of **x**. log returns the natural logarithm of x; **log10** returns the base 10 logarithm. **Pow** returns $\mathbf{x}^{\mathbf{y}}$. **sqrt** returns the square root of **x**.

A.1.3 Hyperbolic Functions: sinh, cosh, tanh

```
double sinh(double x)

double cosh(double x)

double tanh(double x)
```

These functions compute the designated hyperbolic functions for real double-precision arguments.

A.1.4 Absolute Value, Floor, Ceiling: abs, fabs, floor, ceil

```
int abs(int i)

double floor(double x)

double ceil (double x)

double fabs(double x)
```

abs returns the absolute value of its integer operand (MATH.H is not required for this integer function). **fabs** returns the absolute value of the double value **x. floor** returns the largest integer not greater than **x. ceil** returns the smallest integer not less than **x**.

A.1.5 Euclidean Distance: `hypot, cabs`

```
double hypot(double x, double y)

double _cabs(struct _complex z);
```

hypot and **cabs** return **sqrt(x*x + y*y)**. The **cabs** requires a **_complex** structure containing two doubles which can be used for representing a complex number.

A.2 CHARACTER STRING FUNCTIONS

The *string functions* allow a program to perform basic string manipulation of null-terminated character strings. They are declared in the header file STRING.H.

A.2.1 Convert String to Double-precision Number: strtod, atof

```
double strtod (char *str, char **endptr)

double atof (char *str)
```

strtod returns a double-precision floating point number represented by the character string pointed to by **str**. The string is scanned up to the first unrecognized character. **strtod** recognizes an optional string of white space characters, an optional sign, a string of digits optionally containing a decimal point, an optional e or E followed by an optional sign or space, followed by an integer. If the value of **endptr** is not NULL, a pointer to the character terminating the scan is returned in the location pointed to by **endptr.** If no number can be formed, ***endptr** is set to **str,** and zero is returned. **atof(str)** is equivalent to **strtod(str, (char **)NULL)**.

A.2.2 Convert String to Integer: `strtol, atol, atoi`

```
long strtol (char *str, char **endptr, int base)

long atol (char *str)

int atoi (char *str)
```

strtol returns a long integer represented by the character string **str**. The string is scanned up to the first character inconsistent with the base. Leading white space characters are ignored. If the value of **endptr** is not NULL, a pointer to the character terminating the scan is returned in ***endptr**. If no integer can be formed, ***endptr** is set to **str,** and zero is returned. If base is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and 0x or 0X is ignored if base is 16. If base is zero, the string itself determines the base as follows: After an optional

leading sign, a leading zero indicates octal conversion, and a leading 0x or 0X hexadecimal conversion. Otherwise, decimal conversion is used. **atol(str)** is equivalent to **strol(str,(char **)NULL,10)** and **atoi(str)** is equivalent to **(int)strtol(str,(char **)NULL,10)**.

A.2.3 Number to String Conversion: ecvt, fcvt, gcvt

```
char *ecvt(double value, int ndigit, int *decpt, int *sign)

char *fcvt(double value, int ndigit, int *decpt, int *sign)

char *gcvt(double value, int ndigit, char *buf)
```

ecvt converts the value to a null-terminated string of ndigit ASCII digits and returns a pointer the string created. The position of the decimal point relative to the beginning of the string is stored indirectly through **decpt** (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by **sign** is non-zero, otherwise it is zero. **fcvt** is identical to **ecvt,** except that the correct digit has been rounded for FORTRAN F format output of the number of digits specified by **ndigits.**

gcvt converts the value to a null-terminated ASCII string in **buf** and returns a pointer to **buf** (previously allocated by the caller). It attempts to produce **ndigit** significant digits in Fortran F format if possible, otherwise E format, ready for printing. Trailing zeros may be suppressed.

A.2.4 String Manipulation Functions: `strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok`

```
char *strcat(char *s1, char *s2)

char *strncat(char *s1, char *s2, int n)

int strcmp(char *s1, char *s2)

int strncmp(char *s1, char *s2, int n)

char *strcpy(char *s1, char *s2)

char *strncpy(char *s1, char *s2, int n)

int strlen(char *s)

char *strchr (char *s, char c)

char *strrchr(char *s, char c)

char *strpbrk(char *s1, char *s2)

int strspn(char *s1, char *s2)
```

```
int strcspn(char *s1, char *s2)

char *strtok(char *s1, char *s2)
```

The arguments **s1, s2,** and s point to strings (arrays of characters terminated by a null character). The functions **strcat, strncat, strcpy,** and **strncpy** all alter **s1.** These functions do not check for overflow of the array pointed to by **s1.**

strcat appends a copy of string **s2** to the end of string **s1.** strncat copies at most **n** characters. Both return a pointer to the null-terminated result.

strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according to if **s1** is lexicographically greater than, equal to, or less than **s2**. **strncmp** makes the same comparison but compares at most the first **n** characters.

strcpy copies string **s2** to **s1,** stopping after the null character has been copied. **strncpy** copies exactly **n** characters, truncating **s2** or adding null characters to **s1** if necessary. The result will not be null terminated if the length of **s2** is **n** or more. Each function returns **s1.**

strlen returns the number of characters in **s,** not including the terminating null character.

strchr returns a pointer to the first occurrence of character **c** in string **s,** or a NULL pointer if **c** does not occur in the string. The null character terminating a string is considered to be part of the string. **strrchr** works the same as **strchr** but returns a pointer to the last occurrence of character **c** in the string.

strpbrk returns a pointer to the first occurrence in string **s1** of any character from string **s2,** or a NULL pointer if no character from **s2** exists in **s1.**

strspn returns the length of the initial segment of string **s1** which consists entirely of characters from string **s2. strcspn** works the same as **strspn** except that the length returned is the length of **s1** which is not in string **s2.**

strtok considers the string **s1** to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string **s2.** The first call returns a pointer to the first character of the first token, and will have written a null character into **s1** immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string **s1** immediately following that token. In this way, subsequent calls will work through the string **s1** until no tokens remain. The separator string **s2** may be different from call to call. When no token remains in **s1,** a NULL pointer is returned.

A.3 MEMORY ALLOCATION OPERATORS

The memory allocation operators in C++ provide dynamic access to all the available memory (called the *free-store*) which can be assigned to a program. **new** allocates a type or an array of a type and **delete** places the memory allocated back in the free-store. The general usage of **new** and **delete** are as follows:

```
Type *px = new Type;
delete px;
```

Or for arrays:

```
Type *px = new Type[NumberToAllocate];
delete [] px;
```

new and **delete** provide a simple general purpose memory allocation package. **new** returns a pointer to the first element in the array or NULL if the allocation is unsuccessful**.** The argument to **delete** is a pointer to a block previously allocated by **new**; this space is made available for further allocation. Needless to say, grave disorder will result if the space assigned by **new** is overrun or if some random number is handed to **delete**.

It is acceptable to pass NULL to delete so no explicit check is needed before freeing memory:

```
// Not necessary
if( px != NULL )
    delete [] px;
// You can do this safely
delete [] px;
```

A.4 STANDARD INPUT/OUTPUT CLASSES

The standard input and output classes described here allow access to the console (keyboard and display device) via the special files (called streams) **cin** and **cout**. The **fstream** class allows a general file (previously opened using **fstream::open**) to be used instead of **cin** or **cout**. The header files IOSTREAM and FSTREAM.H contain these classes.

A.4.1 Get a Character from a Stream: get

```
int istream::get()
```

getc is a member of the istream class which returns the next character from the input stream. These functions return the integer constant EOF at end of file or upon read error.

A.4.2. Get a String from a Stream: getline

```
istream& istream::getline(char *s, int n, char delim)
```

getline reads **n − 1** characters, or up to a delimiter character, whichever comes first, from the stream into the string **s.** The last character read into **s** is followed by a null character. If **delim** is not specified, the default line delimiter is a newline. It returns the reference to the istream class read from.

A.4.3. Get a Block of data from a Stream: read

```
istream& istream::read(char *s, int n)
```

read reads **n** characters from the stream into the memory allocated to **s**. It returns the reference to the istream class read from.

A.4.4. Get a Class from a Stream: `operator >>`

```
istream& operator>>(Type& t)
```

Classes can define an operator which can read their data directly from an input stream. Inside the **>>** operator, the class can then use **get, getline, read** and other **>>** operators to load the data from the input stream. The **>>** operator has been defined for all standard classes. It returns the reference to the istream class read from.

A.4.5 Send a Character to a Stream: put

```
ostream& ostream::put(char c)
```

put appends the character **c** to the named output stream. It returns the reference to the ostream class written to.

A.4.6 Send a String or Block of data to a Stream: `write`

```
ostream& ostream::write(char *s, int n)
```

write copies **n** bytes of data from the memory pointed to by **s** to the output stream. It returns the reference to the ostream class written to.

A.4.7 Open a File: open

```
void fstream::open(const char* name, int mode, int prot)
```

open opens the file named by the string **name** and associates a stream with it. **mode** is an integer containing ios enumerators which can be combined with the OR (|) operator. The ios enumerators are defined constant having one of the following values:

ios::app	Bytes are appended to the end of the file regardless of the file pointer.
ios::ate	Bytes are appended to the end of the file, but the file pointer can be moved.
ios::in	The file is opened for input. The original file (if it exists) will not be truncated.

ios::out	The file is opened for output.
ios::trunc	If the file already exists, its contents are discarded. This mode is implied if **ios::out** is specified, and **ios::ate**, **ios::app**, and **ios:in** are not specified.
ios::nocreate	If the file does not already exist, the function fails.
ios::noreplace	If the file already exists, the function fails.
ios::binary	Opens the file in binary mode (the default is text mode).

A.4.8 Determine the Position of a File Pointer: `tellg, tellp`

```
streampos fstream::tellg()

streampos fstream::tellp()
```

The member functions **tellg** and **tellp** return the positions (in bytes) from the beginning of the file of the read and write pointers respectively. The **streampos** type is defined in IOSTREAM.H.

A.4.9 Resposition a File Pointer: `seekg, seekp`

```
istream& fstream::seekg(streampos pos)
istream& fstream::seekg(streamoff off, int direction)

ostream& fstream::seekp(streampos pos)
ostream& fstream::seekp(streamoff off, int direction)
```

The member functions **seekg** and **seekp** set the read and write pointers respectively. The first version of each function sets the pointer to the **pos** parameter which is the position (in bytes) from the beginning of the file. The second version of each function sets the pointer to **off** bytes from the **direction**. The **direction** parameter can be **ios::beg**, **ios::cur**, or **ios::end** which are the beginning of the file, the current file pointer, or the end of the file respectively.

A.4.10 Close a File: close

```
void fstream::close()
fflush(FILE *stream)
```

fclose causes any buffers associated with the named stream to be emptied, and the file closed. **fclose** is performed automatically upon calling **exit. fflush** causes any buffered data for the named output stream to be written to that file. The stream remains open. These routines return the constant EOF (usually -1) if stream is not associated with an output file, or if buffered data cannot be transferred to that file.

A.4.11 Formatted Output Conversion: `printf, sprintf`

```
int printf(char *format ;ob, arg ;cb ... )

int sprintf(char *s, char *format ;ob, arg ;cb ... )
```

printf places output on the standard output stream **stdout. sprintf** places output in the string **s,** followed by the null character.

Both of these functions converts, formats, and prints its arguments under the control of the control string, **format**. The first argument is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument.

Each conversion specification is introduced by the character **%.** Following the **%,** there may be the following optional characters:

- Zero or more flags, which modify the meaning of the conversion specification.
- An optional minus sign, **-,** which specifies left adjustment of the converted value in the indicated field.
- An optional digit string specifying a field width.
- An optional period **.** which serves to separate the field width from the next digit string.
- An optional digit string specifying a precision which specifies the number of digits to appear after the decimal point, for **e** and **f** conversion, or the maximum number of characters to be printed from a string.
- The character 1 specifying that a following **f, d, o, x,** or **u** corresponds to a long type.
- A character which indicates the type of conversion to be applied.
- A field width or precision may be * instead of a digit string (in this case the integer argument following the converted value supplies the field width or precision).

The flag characters and their meanings are as follows:

-	The result of the conversion will be left justified within the field.
+	The result of a signed conversion will always begin with a sign (+ or -).
Blank	If the first character of a signed conversion is not a sign, a blank will be prepended to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.
#	The value is to be converted to an alternate form. For **c, d, s** and **u** conversions, the flag has no effect. For **o** conversions, it increases the precision to force the first digit of the result to be a zero. For **x** or **X** conversions, a non-zero result will have 0x or 0X prepended to it. For **e, E, f, g** and **G**

conversions, the result will always contain a decimal point, even if no digits follow the point. A decimal point usually appears in the result of these conversions only if a digit follows it.

The conversion characters and their meanings are as follows:

d, o, x The integer **arg** is converted to decimal, octal, or hexadecimal notation respectively.

f The **float** or **double arg** is converted to decimal notation in the style '[-]ddd.ddd' where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is 0, no digits and no decimal point are printed.

e The **float** or **double arg** is converted in the style `[-]d.ddde+dd' where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced.

g The **float** or **double arg** is printed in style **f,** or in style **e.** The style used depends on the value to be printed: style **e** will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeroes are removed from the result. A decimal point appears only if it is followed by a digit.

c The character **arg** is printed.

s **arg** is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however if the precision is 0 or missing all characters up to a null are printed.

u The unsigned integer **arg** is converted to decimal and printed.

% Print a **%;** no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. **printf** and **sprintf** return the number of characters transmitted, not including the null in the case of **sprintf** or a negative value if an output error was encountered. **printf** displays the results of a program by sending formatted output to the display device (terminal or dedicated alphanumeric display screen).

The format string used with the **printf** function contains the text to be printed and the format to display the variables listed as arguments to **printf**. **printf** may have any number of arguments but the number of arguments must equal the number of items declared in the format string. For example, the following **printf** statement prints three variables:

```
printf("\nAve = %f Var = %8.2f N = %d",ave,var,n);
```

The variables **ave** and **var** must be declared as **float** variables and **n** must be an **int**. The three parts of the format string which define the display formats of the variables (called format specifiers) are:

1. **%f** for a standard floating display of **ave;**
2. **%8.2f** for a floating point display of **var** with a width of eight and two places after the decimal point; and
3. **%d** for a standard integer display of **n**.

The format specifiers which can be used with **printf** are as follows (**[ ]** indicates optional fields):

%[w]d	signed integer, width **w**
%[w]p	pointer value, width **w**
%[w.d]f	floating point, width **w**, **d** places after decimal point
%[w.d]e	floating point in exponential format, width **w**,**d** places after decimal point
%c	single character
%s	string
%[w]x	integer in hexadecimal format, lower case, width **w**
%[w]X	integer in hexadecimal format, upper case, width **w**

There are several special escape sequences which may be used in the format string for special control of the display device as follows:

%%	prints a single percent sign (%).
****	prints a single backslash (\).
\’	prints an apostrophe (‘).
\”•	prints a quotation mark (“).
\n	generates a newline (carriage return and linefeed).
\t	generates a horizontal tab.
\b	generates a backspace.
\f	generates a formfeed or clear screen.
\r	generates a carriage return.
\000	defines a character represented by ASCII code **000**, where **000** is 1 to 3 octal digits (0-7).
\xHHH	prints a character represented by ASCII code **HHH**, where **HHH** is 1 to 3 hexadecimal digits (0-9,A-F).

A.4.12 Formatted Input Conversion: `scanf, sscanf`

```
int scanf(char *format ;ob , pointer ;cb ... )
int sscanf(char *s, char *format ;ob , pointer ;cb ... )
```

scanf "scans" a line entered by the user and generates data in a number of variables which are arguments to the function. scanf reads from the standard input stream stdin. sscanf "scans" a string s. Both functions read characters, interpret them according to a format, and store the results in the list of arguments. Each expects, as arguments, a control string format described below, and a set of pointer arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain the following:

White space characters (blanks, tabs, new lines, or form feeds) which, except in two cases described below, cause input to be read up to the next non-white space-character.

An ordinary character (not **%**), which must match the next character of the input stream.

Conversion specifications, consisting of the character **%,** an optional assignment suppressing character *, an optional numerical maximum field width, an optional **l** or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except [ ] and **c,** white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

%	a single **%** is expected in the input at this point; no assignment is done.
d	a decimal integer is expected; the corresponding argument should be an integer pointer.
u	an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.
o	an octal integer is expected; the corresponding argument should be an integer pointer.
x	a hexadecimal integer is expected; the corresponding argument should be an integer pointer.
e,f,g	a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer

to a **float.** The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an E or an e, followed by n optional +, -, or space, followed by an integer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating null, which will be added automatically. The input field is terminated by a white-space character.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use **%ls.** If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[,] indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which is called the scanset, and a right bracket. The input field must be a sequence of characters described by the scanset. The circumflex **(^),** when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters not contained in the remainder of the scanset string. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating null, which will be added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters **d, u, o,** and **x** may be capitalized or preceded by **l** or **h** to indicate that a pointer to long or to short rather than to **int** is in the argument list. Similarly, the conversion characters **e, f,** and **g** may be capitalized or preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list. The **l** or **h** modifier is ignored for other conversion characters.

scanf conversion terminates at end of file, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream. **scanf** returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, the constant EOF (usually -1) is returned. All three functions return EOF on end of input and a short count for missing or illegal data items.

A.5 OTHER STANDARD FUNCTIONS

A.5.1 Random Number Generator: rand, srand

```
srand(int seed)

int rand()
```

rand uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the integer range. The generator is reinitialized by calling **srand** with 1 as the seed argument. It can be set to a random starting point by calling **srand** with a different seed argument.

A.5.2 Quick General Sort: qsort

```
qsort(char *base, int nel, int width, int (*compar)())
```

qsort is an implementation of the quick sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine to be called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according to if the first argument is to be considered less than, equal to, or greater than the second.

A.5.3 Terminate a Process and Close Files: exit

```
void exit(int status)
```

The function **exit** terminates a calling process with the following consequences:

1. All of the file descriptors open in the calling process are closed.
2. The parent process of the calling process is notified of the calling process's termination and the lower eight bits of status are made available to the parent process.

Typically, a status value of zero is used to indicate a normal termination, and a non-zero value indicates some error.

DSP Function Library and Programs

This appendix describes the basic use of the DSP library functions and programs provided on the accompanying disk. All the routines described here are provided as source code on the diskette. Full listings of many of the programs are provided in the specific chapters describing the various DSP topics. Section B.1 gives the functions which should be incorporated into a library on the machine the programs are going to be compiled. A linker can then be used to select the object modules from the library required by each program. By using a library, the use of the DSP functions is greatly simplified. A library manager program (often provided with the compiler for a system) should be used to create the library. Section B.2 gives a brief description of each program contained on the disk. In both Sections B.1 and B.2, the relevant sections of the chapters which describe the routines in more detail are indicated.

B.1 LIBRARY FUNCTIONS

The following table indicates the name of each function, the source file name, a short description of its purpose, and the section in the text where more information is located:

Source Filename	Function Name	Text Section	Description
GET.H	getInput	3.1.2	get string/number with prompt
DISKIO.CPP	DSPFile.isFound	3.2.1	determine if DSP file exists
DISKIO.CPP	DSPFile.openRead	3.2.1	open DSP data file to be read

```
DISKIO.CPP  DSPFile.openWrite    3.2.1    open DSP data file for write
DISKIO.CPP  DSPFile.close        3.2.2    close DSP data file
DISKIO.CPP  DSPFile.seek         3.2.2    move arround within DSP file
DISKIO.CPP  DSPFile.getTrailer   3.2.3    read the trailer text
DISKIO.CPP  DSPFile.setTrailer   3.2.3    write the trailer text
DISK.H      DSPFile.readElement  3.2.2    read one data element from disk
DISK.H      DSPFile.writeElemnet 3.2.2    write one data element to disk
DISK.H      DSPFile.read         3.2.2    read vector or matrix from disk
DISK.H      DSPFile.write        3.2.2    write vector or matrix to disk
FILTER.H    Filter.FIRFilter     4.2.1    FIR filter data
FILTER.H    Filter.IIRFilter     4.3.1    IIR filter data
FILTER.H    Filter.filterElement 4.3.1    IIR filter data
FILTER.H    Filter.getHistory    4.3.1    FIR filter get history data
FILTER.H    Filter.reset         4.3.1    set history data to zero
DSP.H       gaussian             4.7.1    Generate Gaussian noise
DSP.H       uniform              4.7.1    Generate uniform noise
DSP.H       min                  4.7.1    Find minimum of two types
DSP.H       max                  4.7.1    Find maximum of two types
DSP.H       round                4.2.2    Find maximum of two types
DSP.H       log2                 5.8.1    Find integer log base 2

DFT.CPP     dft                  5.1    Discrete Fourier transform
DFT.CPP     idft                 5.2    Inverse DFT
DFT.CPP     fft                  5.3    Decimation in time FFT
DFT.CPP     ifft                 5.4    DIT inverse FFT
DFT.CPP     rfft                 5.7.1    Trig recombination real FFT
DFT.CPP     ham                  5.5    Hamming window
DFT.CPP     han                  5.5    Hanning window
DFT.CPP     triang               5.5    Triangle window
DFT.CPP     black                5.5    Blackman window
DFT.CPP     harris               5.5    4 term Blackman-Harris window

MATRIX.H    Matrix.<<            6.2    print the elements of a Matrix
MATRIX.H    Matrix.scale         6.2.1    scale all of a Matrix
MATRIX.H    Matrix.conv          6.2.1    Convert one matrix type to another
MATRIX.H    Matrix.transpose     6.2.1    transpose a matrix
MATRIX.H    Matrix.add           6.2.2    add two matrices
MATRIX.H    Matrix.sub           6.2.2    subtract two matrices
MATRIX.H    Matrix.pwisemult     6.2.2    multiply element by element
MATRIX.H    Matrix.mult          6.2.2    Matrix cross product
MATRIX.H    Matrix.invert        6.2.3    invert a square matrix
MATRIX.H    Matrix.det           6.2.4    determinant of square matrix

IMAGE.CPP   dct2d                7.1.4    Discrete Cosine transform in 2D
IMAGE.CPP   idct2d               7.1.4    Inverse DCT in 2D
IMAGE.CPP   histogram            7.2.1    Finds the histogram of a matrix
IMAGE.CPP   convol2d             7.3.2    Convolve a matrix with a filter
IMAGE.CPP   nonlin2d             7.4    Nonlinear filter (min, max, median)
```

Each of the following sections gives a brief description of the function along with its calling parameters and the function prototype.

B.1.1 User Interface Functions

The user input functions can be found in the GET.H include file. The prototypes for all of the user input functions are as follows:

```
////////////////////////////////////////////////////////////////////////////
//
// getInput( const char *msg, String &input )
//  Get a string from the user.
//
// Throws:
//  DSPException
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// template <class Type>
// getInput( const char *msg, Type &input, Type rangeMin, Type rangeMax )
//  Continue displaying message msg until user
//  gives valid Type input.
//
// Throws:
//  DSPException
//
////////////////////////////////////////////////////////////////////////////
```

B.1.2 Disk Storage Functions

The prototypes for all of the disk storage functions can be found in the DISK.H include file along with the structure definitions required to use the functions. The prototypes, class definitions defined in file DISK.H are as follows:

```
////////////////////////////////////////////////////////////////////////////
//
// convType( const type_info &ti )
//  Converts C++ run-time type information to C DSP_FILE
//
// Returns:
//  Enumerated C DSP_FILE type (see DSPFILETYPE above)
//
////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////
//
// convBuffer( Type *pto, void *pfrom, DSPFILETYPE dft, int len )
//  Converts buffer from C DSP_FILE type to generic Type.
//
//  Uses inline helper function CONV to copy
//  and cast from on type to another.
//
// Throws:
//  DSPException
//
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
//
// DSPFile class
//  Information for C DSP_FILE compatible files
//
////////////////////////////////////////////////////////////////////////
class DSPFile
{
private:
    ////////////////////////////////////////////////////////////////////
    //
    // File header compatible with C DSP_FILE
    //
    ////////////////////////////////////////////////////////////////////

    // Data type 0-7 as defined above
    DSPFILETYPE m_type;

    // Size of each element
    unsigned char m_elementSize;

    // Number of records
    unsigned short int m_numRecords;

    // Number of elements in each record
    unsigned short int m_recLen;

    ////////////////////////////////////////////////////////////////////
    //
    // Information about opened file
    //
    ////////////////////////////////////////////////////////////////////

    // Pointer to trailer text
    char *m_trailer;
```

```
    // File data
    fstream m_fs;
    bool m_readOnly;
    streampos m_posBeginData;

    // Buffer for reading single elements
    BYTE *m_singleElement;

    ////////////////////////////////////////////////////////////////////////
    //
    // Helper functions
    //
    ////////////////////////////////////////////////////////////////////////

    // Clear data
    void empty();

public:
    ////////////////////////////////////////////////////////////////////////
    //
    // Constructor and Destructor
    //
    ////////////////////////////////////////////////////////////////////////
    DSPFile();
    ~DSPFile();

    ////////////////////////////////////////////////////////////////////////
    //
    // File operations
    //
    ////////////////////////////////////////////////////////////////////////
    static bool isFound( const char *name );
    bool hasData() { return( ( m_numRecords * m_recLen ) != 0 ); }
    void openRead( const char *name );
    void openWrite( const char *name );
    void close();

    ////////////////////////////////////////////////////////////////////////
    //
    // Get file format
    //
    ////////////////////////////////////////////////////////////////////////
    DSPFILETYPE getType() { return m_type; }
    unsigned short getNumRecords() { return m_numRecords; }
    unsigned short getRecLen() { return m_recLen; }

    ////////////////////////////////////////////////////////////////////////
    //
    // Get and Set file contents
```

```
//
////////////////////////////////////////////////////////////////////////
void getTrailer( String& str );
void setTrailer( const char *trailer );
void seek( unsigned short len = 0, unsigned short rec = 0 );

////////////////////////////////////////////////////////////////////////
//
// Read and Write templated types
//
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
//
// readElement( Type& element )
//  Reads single element from file.
//
// Note:
//  Can only be used on files with one record.
//
// Throws:
//  DSPException
//
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
//
// writeElement( const Type& element )
//  Writes single element to file.
//
// Note:
//  Can only be used on files with no data or single records.
//
// Throws:
//  DSPException
//
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
//
// read( Vector<Type>& vec )
//  Reads vector from file. Converts from C DSP_FILE type
//  to type of vector.
//
// Throws:
//  DSPException
//
////////////////////////////////////////////////////////////////////////
```

```
    ////////////////////////////////////////////////////////////////////////
    //
    // write( const Vector<Type>& vec )
    //  Writes vector to file. File will be the same type
    //  as the vector.
    //
    // Throws:
    //  DSPException
    //
    ////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////
    //
    // read( Matrix<Type>& mat )
    //  Reads matrix from file. Converts from C DSP_FILE type
    //  to type of matrix.
    //
    // Throws:
    //  DSPException
    //
    ////////////////////////////////////////////////////////////////////////

    ////////////////////////////////////////////////////////////////////////
    //
    // write( const Matrix<Type>& mat )
    //  Writes Matrix to file. Overwrites previous data
    //  in file if any. File will be the same type
    //  as the matrix.
    //
    // Throws:
    //  DSPException
    //
    ////////////////////////////////////////////////////////////////////////
```

B.1.3 Filter Functions

The prototypes for all of the filter functions can be found in the FILTER.H include file. The structure definitions required to define several FIR filters are contained in the file FIRCOEFS.H as follows:

```
////////////////////////////////////////////////////////////////////////
//
// 35 point lowpass FIR filter cutoff at 0.2
//
////////////////////////////////////////////////////////////////////////
float FIRLPF35[] =
{
    -6.849167e-003F, 1.949014e-003F, 1.309874e-002F, 1.100677e-002F,
```

```
    -6.661435e-003F, -1.321869e-002F, 6.819504e-003F, 2.292400e-002F,
     7.732160e-004F, -3.153488e-002F, -1.384843e-002F, 4.054618e-002F,
     3.841148e-002F, -4.790497e-002F, -8.973017e-002F, 5.285565e-002F,
     3.126515e-001F, 4.454146e-001F, 3.126515e-001F, 5.285565e-002F,
    -8.973017e-002F, -4.790497e-002F, 3.841148e-002F, 4.054618e-002F,
    -1.384843e-002F, -3.153488e-002F, 7.732160e-004F, 2.292400e-002F,
     6.819504e-003F, -1.321869e-002F, -6.661435e-003F, 1.100677e-002F,
     1.309874e-002F, 1.949014e-003F, -6.849167e-003F
};
FIRFilter<float> FIRLPF( FIRLPF35, ELEMENTS( FIRLPF35 ) );

////////////////////////////////////////////////////////////////////////
//
// 35 point highpass FIR filter cutoff at 0.3 same as fir_lpf35
// except that every other coefficient has a different sign
//
////////////////////////////////////////////////////////////////////////
float FIRHPF35[] =
{
     6.849167e-003F, 1.949014e-003F, -1.309874e-002F, 1.100677e-002F,
     6.661435e-003F, -1.321869e-002F, -6.819504e-003F, 2.292400e-002F,
    -7.732160e-004F, -3.153488e-002F, 1.384843e-002F, 4.054618e-002F,
    -3.841148e-002F, -4.790497e-002F, 8.973017e-002F, 5.285565e-002F,
    -3.126515e-001F, 4.454146e-001F, -3.126515e-001F, 5.285565e-002F,
     8.973017e-002F, -4.790497e-002F, -3.841148e-002F, 4.054618e-002F,
     1.384843e-002F, -3.153488e-002F, -7.732160e-004F, 2.292400e-002F,
    -6.819504e-003F, -1.321869e-002F, 6.661435e-003F, 1.100677e-002F,
    -1.309874e-002F, 1.949014e-003F, 6.849167e-003F
};
FIRFilter<float> FIRHPF( FIRHPF35, ELEMENTS( FIRHPF35 ) );

////////////////////////////////////////////////////////////////////////
//
// 52 point bandpass matched FIR filter for pulse demo
//
////////////////////////////////////////////////////////////////////////
float FIRPULSE52[] =
{
    -1.2579e-002F, 2.6513e-002F, -2.8456e-016F, -5.8760e-002F,
     7.7212e-002F, -1.4313e-015F, -1.1906e-001F, 1.4253e-001F,
    -3.7952e-015F, -1.9465e-001F, 2.2328e-001F, -7.7489e-015F,
    -2.8546e-001F, 3.1886e-001F, -7.8037e-015F, -3.8970e-001F,
     4.2685e-001F, -6.3138e-015F, -5.0365e-001F, 5.4285e-001F,
    -1.2521e-014F, -6.2157e-001F, 6.6052e-001F, -9.0928e-015F,
    -7.3609e-001F, 7.7207e-001F, -3.6507e-015F, -8.3886e-001F,
     8.6905e-001F, -1.1165e-014F, -9.2156e-001F, 9.4336e-001F,
    -3.8072e-015F, -9.7694e-001F, 9.8838e-001F, -1.1836e-014F,
    -9.9994e-001F, 9.9994e-001F, -3.3578e-015F, -9.5304e-001F,
```

```
     8.9670e-001F, -9.3099e-015F, -7.3609e-001F, 6.4111e-001F,
    -5.9892e-015F, -4.4578e-001F, 3.5365e-001F, -2.8959e-015F,
    -1.9465e-001F, 1.3058e-001F, -1.7477e-016F, -3.4013e-002F
};
FIRFilter<float> FIRPULSE( FIRPULSE52, ELEMENTS( FIRPULSE52 ) );
```

The structure definitions required to define two IIR filters are contained in the file IIRCOEFS.H as follows:

```
////////////////////////////////////////////////////////////////////////
//
// IIR lowpass 3 section (5th order) elliptic filter
// with 0.28 dB passband ripple and 40 dB stopband attenuation.
// The cutoff frequency is 0.25*fs.
//
////////////////////////////////////////////////////////////////////////
float IIRLPF5[] =
{
     0.0552961603F,
    -0.4363630712F, 0.0000000000F, 1.0000000000F, 0.0000000000F,
    -0.5233039260F, 0.8604439497F, 0.7039934993F, 1.0000000000F,
    -0.6965782046F, 0.4860509932F, -0.0103216320F, 1.0000000000F
};
IIRFilter<float> IIRLPF( IIRLPF5, ELEMENTS( IIRHPF6 ), 3 );

////////////////////////////////////////////////////////////////////////
//
// IIR highpass 3 section (6th order) chebyshev filter
// with 1 dB passband ripple and cutoff frequency of 0.3*fs.
//
////////////////////////////////////////////////////////////////////////
float IIRHPF6[] =
{
     0.0025892381F,
     0.5913599133F, 0.8879900575F, -2.0000000000F, 1.0000000000F,
     0.9156184793F, 0.6796731949F, -2.0000000000F, 1.0000000000F,
     1.3316441774F, 0.5193183422F, -2.0000000000F, 1.0000000000F
};
IIRFilter<float> IIRHPF( IIRHPF6, ELEMENTS( IIRHPF6 ), 3 );
```

B.1.4 DFT Functions

The prototypes for all of the DFT functions can be found in the DFT.H include file along with the definition of the **COMPLEX** structure required for many of the functions. The prototypes in the file DFT.H is as follows:

```
////////////////////////////////////////////////////////////////////////
//
// Fourier Transform functions
//
////////////////////////////////////////////////////////////////////////
template <class Complex>
Vector<Complex> fft( const Vector<Complex>& vIn, int lenFFT );
template <class Complex>
Vector<Complex> ifft( const Vector<Complex>& vIn, int lenFFT );
template <class Complex>
Vector<Complex> dft( const Vector<Complex>& vIn );
template <class Complex>
Vector<Complex> idft( const Vector<Complex>& vIn );
template <class Type>
Vector<Complex> rfft( const Vector<Type> vIn );
template <class Complex>
Vector<Complex> ham( const Vector<Complex>& vIn );
template <class Complex>
Vector<Complex> han( const Vector<Complex>& vIn );
template <class Complex>
Vector<Complex> triang( const Vector<Complex>& vIn );
template <class Complex>
Vector<Complex> black( const Vector<Complex>& vIn );
template <class Complex>
Vector<Complex> harris( const Vector<Complex>& vIn );
```

B.1.5 Matrix Functions

The prototypes for all of the matrix functions can be found in the MATRIX.H include file along with the definition of the **Matrix** class required for many of the functions. The file DISK.H includes classes which can read and write the **Matrix** class. The prototypes in the file MATRIX.H are as follows:

```
////////////////////////////////////////////////////////////////////////
//
// Matrix<Type> add( const Matrix<Type>& m1, const Matrix<Type>& m2 )
//  Add two matrices.
//
// Throws:
//  DSPException
//
// Returns:
//  Resultant matrix ( m1 + m2 )
//
////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////
//
// Matrix<Type> sub( const Matrix<Type>& m1, const Matrix<Type>& m2 )
//  Subtract two matrices.
//
// Throws:
//  DSPException
//
// Returns:
//  Resultant matrix ( m1 - m2 )
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// Matrix<Type> pwisemult( const Matrix<Type>& m1, const Matrix<Type>& m2 )
//  Multiply elements of two matrices.
//
// Throws:
//  DSPException
//
// Returns:
//  Resultant matrix ( m1 * m2 )
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// Matrix<Type> mult( const Matrix<Type>& m1, const Matrix<Type>& m2 )
//  Cross product of two matrices.
//
// Throws:
//  DSPException
//
// Returns:
//  Resultant matrix ( m1 cross m2 )
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// Matrix<Type> transform(
//  const Matrix<Type>& m1,
//  const Matrix<Type>& t1,
//  const Matrix<Type>& t2 )
//  Transform matrix by multiplying submatrices by transform blocks.
//
// Throws:
```

```
//  DSPException
//
// Returns:
//  Resultant matrix ( t1 * m1 * t2 )
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// Matrix<Type> transpose( const Matrix<Type>& m )
//  Transpose elements of a matrix.
//
// Throws:
//  DSPException
//
// Returns:
//  Resultant matrix ( mT )
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// sum( float scale, const Matrix<Type>& m )
//  Sum all elements of a matrix.
//
// Returns:
//  Resultant scalar sum
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// scale( float scale, const Matrix<Type>& m )
//  Scales all elements of a matrix by a float value.
//
// Returns:
//  Resultant matrix (s * m)
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// Matrix<Type1>& conv( Matrix<TypeTo>& mTo, const Matrix<TypeFrom>& mFrom
)
//  Convert a matrix from one type to another.
//
// Throws:
//  DSPException
```

```
//
// Returns:
//  Resultant matrix ( mTo = (mTo's type)mFrom )
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// Matrix<Type> invert( const Matrix<Type>& m )
//  Invert a square matrix.
//
// Throws:
//  DSPException
//
// Returns:
//  Resultant matrix ( mI )
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// double det( const Matrix<Type>& m )
//  Calculate determinant of a matrix.
//
// Throws:
//  DSPException
//
// Returns:
//  Determinant as double.
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// ostream& operator<<( ostream& os, const Matrix<Type>& m )
//  Writes the matrix out to an ostream for display.
//
// Returns:
//  ostream after write
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// Compare matrices
//
////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////
//
// bool operator!=( const Matrix<Type>& m1, const Matrix<Type>& m2 )
//  Compare two matrices element by element.
//
// Throws:
//  DSPException
//
// Returns:
//  true -- Matrices are different
//
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
//
// bool operator==( const Matrix<Type>& m1, const Matrix<Type>& m2 )
//  Compare two matrices element by element.
//
// Throws:
//  DSPException
//
// Returns:
//  true -- Matrices are same
//
////////////////////////////////////////////////////////////////////////
```

B.1.6 Image Processing Functions

The prototypes for all of the image processing functions can be found in the MATRIX.H include file along with the definition of the **MATRIX** class required for many of the functions (see Section B.1.5 for a listing of MATRIX.H). The function prototypes in the IMAGE.CPP source file are as follows:

```
////////////////////////////////////////////////////////////////////////
//
// dct2d( const Matrix<PIXEL>& mIn )
//  Performs the Discrete Cosine Transform (DCT) in two dimensions.
//
// Note:
//  PIXEL is defined in dsptypes.h
//
// Throws:
//  DSPException
//
// Returns:
```

```
//  Matrix<float> of transformed image.
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// idct2d( const Matrix<float>& mIn )
//  Performs the Inverse Discrete Cosine Transform (IDCT)
//  in two dimensions.
//
// Note:
//  PIXEL is defined in dsptypes.h
//
// Throws:
//  DSPException
//
// Returns:
//  Matrix<PIXEL> of transformed image.
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// histogram( const Matrix<PIXEL>& mIn, int min, int max )
//  Calculates histogram of 2D image.
//
// Note:
//  PIXEL is defined in dsptypes.h
//
// Throws:
//  DSPException
//
// Returns:
//  Vector<float>
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// convol2d( const Matrix<int>& mIn, const Matrix<int>& mFilt )
//  Filters image with 2D convolution filter.
//
// Throws:
//  DSPException
//
// Returns:
//  Matrix<int> of convolved image.
//
////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////
//
// convol2d( const Matrix<float>& mIn, const Matrix<float>& mFilt )
//  Filters image with 2D convolution filter.
//
// Throws:
//  DSPException
//
// Returns:
//  Matrix<float> of convolved image.
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// nonlin2d( const Matrix<PIXEL>& mIn, int type, int kernelSize )
//  Filters image with non-linear filter ( erosion, dilation,
//  and median ) on an image matrix of data using a square kernel.
//
// Note:
//  The filter types can be one of:
//   NONLINEROSION
//   NONLINDILATION
//   NONLINMEDIAN
//
//  The kernel size must be odd.
//
// Throws:
//  DSPException
//
// Returns:
//  Matrix<PIXEL> of filtered image.
//
////////////////////////////////////////////////////////////////////////////
```

B.2 PROGRAMS

The following table indicates the name of each program, a short description of its purpose, and the section in the text where more information is located:

Program Name	Section	Description
ADDNOISE.CPP	4.7.1	Add noise to DSP data or create noise
ADNOIS2D.CPP	7.1.1	Add noise to 2D DSP image data
ARFREQ.CPP	4.10.3	AR frequency estimation program
ARMA.CPP	4.10.2	ARMA system modeling program

COMPRESS.CPP	7.1.5	Compress image using the DCT
CON2DFFT.CPP	7.3.4	Convolve using the 2D FFT
COR.CPP	6.1.3	Perform cross correlation or auto correlation
DCTTEST.CPP	7.1.4	Compress image using the DCT
DECIM.CPP	4.5.1	Decimate DSP data
DSP2MAT.CPP	B.2.2	Conversion program from DSP to MATLAB™
DSP2WAV.CPP	B.2.2	Conversion program from DSP to WAV audio
EDETECT.CPP	7.3.5	Edge detection on an image
EXPAND.CPP	7.1.6	Expand compressed image using inverse DCT
FASTCON.CPP	5.9	Fast convolution using the FFT
FFTTEST.CPP	5.8.1	Demonstrate and test FFT and window functions
FIRFILT.CPP	4.2.1	FIR filter the records in a DSP data file
FIRITEST.CPP	4.2.2	FIR filter integer records in a DSP data file
FLATTEN.CPP	7.2.2	Level the histogram of an image
IDCTTEST.CPP	7.1.4	Demonstrate and test Inverse DCT function
IDFTTEST.CPP	5.8.2	Demonstrate and test DFT and IDFT functions
IFFTTEST.CPP	5.8.3	FFT test program
IIRDEZN.CPP	4.3.1	Make IIR filter from analog prototype
IIRFILT.CPP	4.3.2	IIR filter the records in a DSP data file
INTERP.CPP	4.5.1	2:1 and 3:1 FIR filter interpolation
INTFFT.CPP	5.11	Interpolate 2:1 using FFT and IFFT functions
LMS.CPP	4.10.1	Adaptive least mean square algorithm
LSFIT.CPP	6.4.1	Weighted least squares fit using Matrix class
MAKMAT.CPP	6.2	Create matrix of user input data
MAT2DSP.CPP	B.2.2	Conversion program from MATLAB™ to DSP
MATIMAG.CPP	5.10	Scale matrix for image display
MATPRINT.CPP	6.2	Print a matrix
MATTST.CPP	6.2.5	Test matrix routines
MEDIAN.CPP	4.8.2	Conditionally median filter a data record
MKWAVE.CPP	3.3	Generates sum of cosine waves signal
MSEOFPIC.CPP	7.1.7	Find mean squared difference of two images
MULAW.CPP	4.8.3	Speech compression using mu law
NONLIN2D.CPP	7.4.1	Min, max, or median filtering
OSC.CPP	4.9.1	Oscillator program using 2^{nd} order IIR
PSE.CPP	5.10	Power spectral estimation using the FFT
PSHIFT.CPP	4.5.2	Pitch shift or sample rate converter
QUANTIZE.CPP	4.7.4	Quantize a record in a DSP data file
RDFREC.CPP	3.2.2	Read float record and convert
RDRECS.CPP	3.2.2	Read record test program
RDTRAIL.CPP	3.2.3	Read and display trailer
REALCMX.CPP	4.6.1	Convert an RF data record to complex data
REALTIME.CPP	4.4.2	Demonstrate real time filtering
RFFTTEST.CPP	5.8.4	Exercise the rfft function for real FFTs
SMOOTH.CPP	7.3.3	Convolves an image with a 3x3 Gaussian filter
STATS.CPP	4.7.2	Find min, max, mean and variance of records
VECTST.CPP	6.1.2	Test vector macros
WAV2DSP.CPP	B.2.2	Conversion program from WAV audio format to DSP

```
WAVETAB.CPP   4.9.2    Wavetable Sound Generator
WRRECS.CPP    3.1.2    Write record test program
WRTRAIL.CPP   3.2.3    Write trailer test program
```

B.2.1 WINPLOT Program

This program reads DSP data records and plots the samples on an IBM-PC WINDOWS compatible display device. The full source code and the executable files are provided on the disk. The program can also generate a hard copy of the display on the installed printer. The program requires only the filename as an argument. Usage is as follows:

```
WINPLOT <filename>
```

After the plot is completed, the user can move through the records using the up arrow and down arrow keys or print the plot using the menu based interface. Consult the help files and source code on the disk for more information.

B.2.2 File Format Conversion Programs

Four utility programs are provided to allow conversion of the **DSPFile** format to or from the popular WAV audio format and the MATLAB™ matrix file format. The full source code and the executable files are provided on the disk. In each case, the program arguments (two filenames) are supplied by the user on the command line as follows:

DSPFile format to MATLAB™ format:

```
DSP2MAT <DSPinput> <MAToutput>
```

DSPFile format to WAV audio format:

```
DSP2WAV <DSPFileIn> <WAVout>
```

MATLAB™ format to DSPFile format:

```
MAT2DSP <MATinput> <DSPoutput>
```

WAV audio format to DSPFile format:

```
WAV2DSP <WAVinput> <DSPoutput>
```

This page intentionally left blank

Index

LICENSE AGREEMENT AND LIMITED WARRANTY

READ THE FOLLOWING TERMS AND CONDITIONS CAREFULLY BEFORE OPENING THIS DISK PACKAGE. THIS LEGAL DOCUMENT IS AN AGREEMENT BETWEEN YOU AND PRENTICE-HALL, INC. (THE "COMPANY"). BY OPENING THIS SEALED DISK PACKAGE, YOU ARE AGREEING TO BE BOUND BY THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THESE TERMS AND CONDITIONS, DO NOT OPEN THE DISK PACKAGE. PROMPTLY RETURN THE UNOPENED DISK PACKAGE AND ALL ACCOMPANYING ITEMS TO THE PLACE YOU OBTAINED THEM FOR A FULL REFUND OF ANY SUMS YOU HAVE PAID.

1. **GRANT OF LICENSE:** In consideration of your payment of the license fee, which is part of the price you paid for this product, and your agreement to abide by the terms and conditions of this Agreement, the Company grants to you a nonexclusive right to use and display the copy of the enclosed software program (hereinafter the "SOFTWARE") on a single computer (i.e., with a single CPU) at a single location so long as you comply with the terms of this Agreement. The Company reserves all rights not expressly granted to you under this Agreement.

2. **OWNERSHIP OF SOFTWARE:** You own only the magnetic or physical media (the enclosed disks) on which the SOFTWARE is recorded or fixed, but the Company retains all the rights, title, and ownership to the SOFTWARE recorded on the original disk copy(ies) and all subsequent copies of the SOFTWARE, regardless of the form or media on which the original or other copies may exist. This license is not a sale of the original SOFTWARE or any copy to you.

3. **COPY RESTRICTIONS:** This SOFTWARE and the accompanying printed materials and user manual (the "Documentation") are the subject of copyright. You may not copy the Documentation or the SOFTWARE, except that you may make a single copy of the SOFTWARE for backup or archival purposes only. You may be held legally responsible for any copying or copyright infringement which is caused or encouraged by your failure to abide by the terms of this restriction.

4. **USE RESTRICTIONS:** You may not network the SOFTWARE or otherwise use it on more than one computer or computer terminal at the same time. You may physically transfer the SOFTWARE from one computer to another provided that the SOFTWARE is used on only one computer at a time. You may not distribute copies of the SOFTWARE or Documentation to others. You may not reverse engineer, disassemble, decompile, modify, adapt, translate, or create derivative works based on the SOFTWARE or the Documentation without the prior written consent of the Company.

5. **TRANSFER RESTRICTIONS:** The enclosed SOFTWARE is licensed only to you and may not be transferred to any one else without the prior written consent of the Company. Any unauthorized transfer of the SOFTWARE shall result in the immediate termination of this Agreement.

6. **TERMINATION:** This license is effective until terminated. This license will terminate automatically without notice from the Company and become null and void if you fail to comply with any provisions or limitations of this license. Upon termination, you shall destroy the Documentation and all copies of the SOFTWARE. All provisions of this Agreement as to warranties, limitation of liability, remedies or damages, and our ownership rights shall survive termination.

7. **MISCELLANEOUS:** This Agreement shall be construed in accordance with the laws of the United States of America and the State of New York and shall benefit the Company, its affiliates, and assignees.

8. **LIMITED WARRANTY AND DISCLAIMER OF WARRANTY:** The Company warrants that the SOFTWARE, when properly used in accordance with the Documentation, will operate in sub-

stantial conformity with the description of the SOFTWARE set forth in the Documentation. The Company does not warrant that the SOFTWARE will meet your requirements or that the operation of the SOFTWARE will be uninterrupted or error-free. The Company warrants that the media on which the SOFTWARE is delivered shall be free from defects in materials and workmanship under normal use for a period of thirty (30) days from the date of your purchase. Your only remedy and the Company's only obligation under these limited warranties is, at the Company's option, return of the warranted item for a refund of any amounts paid by you or replacement of the item. Any replacement of SOFTWARE or media under the warranties shall not extend the original warranty period. The limited warranty set forth above shall not apply to any SOFTWARE which the Company determines in good faith has been subject to misuse, neglect, improper installation, repair, alteration, or damage by you. EXCEPT FOR THE EXPRESSED WARRANTIES SET FORTH ABOVE, THE COMPANY DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. EXCEPT FOR THE EXPRESS WARRANTY SET FORTH ABOVE, THE COMPANY DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATION REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE.

IN NO EVENT, SHALL THE COMPANY OR ITS EMPLOYEES, AGENTS, SUPPLIERS, OR CONTRACTORS BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR IN CONNECTION WITH THE LICENSE GRANTED UNDER THIS AGREEMENT, OR FOR LOSS OF USE, LOSS OF DATA, LOSS OF INCOME OR PROFIT, OR OTHER LOSSES, SUSTAINED AS A RESULT OF INJURY TO ANY PERSON, OR LOSS OF OR DAMAGE TO PROPERTY, OR CLAIMS OF THIRD PARTIES, EVEN IF THE COMPANY OR AN AUTHORIZED REPRESENTATIVE OF THE COMPANY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL LIABILITY OF THE COMPANY FOR DAMAGES WITH RESPECT TO THE SOFTWARE EXCEED THE AMOUNTS ACTUALLY PAID BY YOU, IF ANY, FOR THE SOFTWARE.

SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT ALWAYS APPLY. THE WARRANTIES IN THIS AGREEMENT GIVE YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY IN ACCORDANCE WITH LOCAL LAW.

ACKNOWLEDGMENT

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU ALSO AGREE THAT THIS AGREEMENT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN YOU AND THE COMPANY AND SUPERSEDES ALL PROPOSALS OR PRIOR AGREEMENTS, ORAL, OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN YOU AND THE COMPANY OR ANY REPRESENTATIVE OF THE COMPANY RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

Should you have any questions concerning this Agreement or if you wish to contact the Company for any reason, please contact in writing at the address below.

Robin Short
Prentice Hall PTR
One Lake Street
Upper Saddle River, New Jersey 07458

ABOUT THE CD-ROM

All the files for the book, *C++ Algorithms for Digital Signal Processing* are in the directory \DSPC on the CD-ROM. The following sub-directories contain the source code and pre-compiled versions of the C++ programs which can be run on computers with a version of Microsoft Windows® (Windows 95® or later, Windows NT® 4.0 or later) operating system:

Sub-directory	**Contents**
CPP	C++ source code for all programs in the book and compiler information for version 5 or 6 of the Microsoft Visual C++ compiler
CPP\DSPPROJ	Project files and folders for all DSP programs in the book
ANSI_C	C versions of DSP programs, REMEZ.C FIR filter program. A full description of the ANSI C code is in file CCODE.HTM (HTML file for use with Internet Explorer or Browser).
BIN	Compiled versions of all executable programs
WINPLOT	C++ source code for WINPLOT DSP data plot program and compiler project information for version 6 of the Microsoft Visual C++ compiler
DATA	Input data files for examples
DATA\SIGNALS	Result data files from 1-D DSP examples
DATA\IMAGES	Result data files from image processing examples

INSTALLING THE DSP FILES

Copy DSP directories from the CD-ROM to your hard disk. For example, enter the following command at the command prompt:

XCOPY /S D:\DSPC*.* C:\DSPC*.*

WILL COPY FILES FROM THE CD-ROM (the drive D:) TO A DIRECTORY CALLED \DSPC ON THE HARD DRIVE (the drive C:).

If you have the Microsoft Visual C++ compiler version 5 or 6, you can open the workspace file \DSPC\CPP\DSPPROJ\DSPPROJ.DSW which contains 54 projects for all of the DSP programs described in the book.

To execute all the precompiled C++ programs referenced in the book, move to the \DSPC\BIN directory and then type the name of the program at the command prompt or double click on the executable file from Windows Explorer. You may wish to put this directory in the PATH so that the programs can be executed from any directory (for example, a directory containing data files). Otherwise, the data files (directory\DSPC\DATA) which are used by the DSP programs must be specified with their full path names.

INSTALLING VISUAL C++ 6.0, INTRODUCTORY VERSION

Insert the CD-ROM. The Installation Wizard should start automatically. If the Wizard does not open automatically, execute the SETUP.EXE program at the root directory of the CD-ROM.

MICROSOFT SOFTWARE

PRODUCT NAME AND ASSOCIATED TRADEMARK: Microsoft® Visual C++® development system

ENVIRONMENT: Microsoft Windows®

SYSTEM REQUIREMENTS

- Personal computer with a 486DX/66 (Pentium 90 or higher microprocessor recommended)
- Microsoft Windows® 95 or later, Windows NT® 4.0, with service pack 3 or later (service pack 3 included)
- Minumum memory: 24 MB for Window 95 or later, 24 MB for Windows NT 4.0 (32 MB recommended for all)
- Hard-disk space required:
  Typical installation: 225 MB
  Maximum installation: 305 MB
- Microsoft Internet Explorer 4.01 Service Pack 1 (included); additional hard disk space required for Microsoft Internet Explorer: 43 MB typical, 59 MB maximum
- CD-ROM drive
- VGA or higher-resolution monitor (Super VGA recommended)
- Microsoft Mouse or compatible pointing device

DOCUMENTATION: On-line help only, no printed documentation

Prentice Hall does not offer technical support for this software. However, if there is a problem with the media, you may obtain a replacement copy by e-mailing us with your problem at: disc_exchange@prenhall.com.