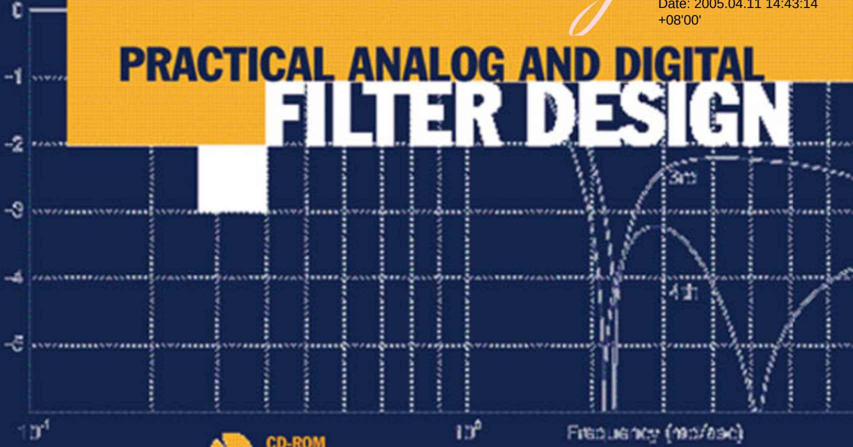
LES THEDE

PRACTICAL ANALOG AND DIGITAL FILTER DESIGN



CD-ROM INCLUDED

Practical
Analog and Digital Filter Design

Artech House, Inc.

Les Thede

2004

This text is dedicated to my wife who keeps me grounded,
and to my grandchildren who know no bounds.

Contents

Preface

This book was intentionally written to be different from other filter design books in two important ways. First, the most common analog and digital filter design and implementation methods are covered in a no-nonsense manner. All important derivations and descriptions are provided to allow the reader to apply them directly to his or her own filter design problem. Over forty examples are provided to help illustrate the fundamentals of filter design. Not only are the details of analog active and digital IIR and FIR filter design presented in an organized and direct manner, but implementation issues are discussed to alert the reader to potential pitfalls. An added feature to this text is the discussion of fast Fourier transforms and how they can be used in filtering applications. The simulation of analog filters is made easier by the generation of PSpice circuit description files that include R-C component values calculated directly from the filter coefficients. In addition, the testing of IIR and FIR filters designed for audio signals is enhanced by providing sample sound files that can be filtered by using the digital filter design coefficients. Anyone with a sound card on their computer can then play the original and processed sound files for immediate evaluation.

The second difference between this book and others is that the text is accompanied by WFilter, a fully functional, Windows®-based filter design software package, and the source code on which it is based. The CD provides the reader with the ability to install WFilter with a few simple clicks of the mouse, and also supplies the reader with the well organized and clearly documented source code detailing the intricacies of filter design. No, the source code provided is not just a collection of fragmented functions, but rather a set of three organized programs that have been developed (with the addition of an easy-to-use graphical interface) into the organized structure of WFilter.

A basic knowledge of C programming is expected of the reader, but the code presented in the text and the appendixes is thoroughly discussed and well documented. The text does assume the reader is familiar with the fundamental concepts of linear systems such as system transfer functions and frequency response although no prior knowledge of filter design is needed.

CHAPTER CONTENTS

Chapter 1 introduces the reader to the filter design problem. An overview of WFilter is presented. Chapter 2 develops the normalized transfer functions for the Butterworth, Chebyshev, inverse Chebyshev, and elliptic approximation cases. Chapter 3 describes the conversion of the normalized lowpass filter to an unnormalized lowpass, highpass, bandpass, or bandstop filter. In addition, the calculation of the frequency response for analog filters is discussed. By the end of the third chapter, a complete analog filter design can be performed. In Chapter 4, the implementation of analog filters is considered using popular techniques in active filter design with discussion of real-world considerations. A PSpice circuit description file is generated to enable the filter developer to analyze the circuit. Chapter 4 completes the discussion of analog filters in this book.

Chapter 5 begins the discussion of discrete-time systems and digital filter design in this book. Several key features of discrete-time systems, including the notion of analog-to-digital conversion, Nyquist sampling theorem, the *z*-transform, and discrete-time system diagrams, are reviewed. Similarities and differences between discrete-time and continuous-time systems are discussed. In Chapter 6, digital IIR (recursive) filters are designed. Three methods of designing IIR filters are considered. In addition, the frequency response calculations and related C code for the IIR filter are developed. Chapter 7 considers digital FIR (nonrecursive) filters using a variety of window methods and the Parks-McClellan optimization routine. The special techniques necessary for FIR frequency response calculation are discussed. The implementation of real-time and nonreal-time digital FIR and IIR filters is discussed in Chapter 8. Implementation issues such as which type of digital filter to use, accuracy of quantized samples, fixed or floating point processing, and finite register length computation are discussed. The reader can then hear the effects of filtering by replaying the original and processed sound files on a sound card. Chapter 9 completes the text with an introduction of the discrete Fourier transform and the more efficient fast Fourier transform (FFT). The reader will learn how to use the FFT in filtering applications and see the code necessary for this operation.

For those readers who desire filter design references or further details of the C code for the design of analog and digital filters, nine separate appendixes provide that added information.

ACKNOWLEDGMENTS

I would not have been able to complete this book without the help and support of a number of people. First, I thank the reviewers of this text who provided many helpful comments, both in the initial and final stages of development. In particular I want to thank Walter A. Serdijn of Delft University of Technology, The Netherlands.

I also thank the friendly people at Artech House, Inc. who have provided me with so much help. This book could not have been written without their professional guidance throughout the publication process.

I also thank Ohio Northern University and the Department of Electrical & Computer Engineering and Computer Science for their support.

And finally, I thank my wife Diane for all of her encouragement and for the many hours of proofreading a second text that made no sense to her!

TRADEMARKS

Windows® is a registered trademark of Microsoft Corp.

Chapter 1

Introduction to Filters and Filter Design Software

Everyone has probably come in contact with one type of filter or another in their lifetime. Maybe it was a coffee filter used to separate the grounds from the liquid, or perhaps an oil filter to remove contaminants from the oil of an engine. Anyone working in an office often filters the unimportant work from the important. In essence then the act of filtering is the act of separating desired items from undesired items. Of course when we discuss filters in this text, we are not talking about coffee, oil, or paperwork, but rather electronic signals. The electronic filters we will be designing will separate the desirable signal frequencies from the undesirable, or in other applications simply change the frequency content which then changes the signal waveform.

There are many types of electronic filters and many ways that they can be classified. A filter's frequency selectivity is probably the most common method of classification. A filter can have a lowpass, highpass, bandpass, or bandstop response, where each name indicates how a band of frequencies is affected. For example, a lowpass filter would pass low frequencies with little attenuation (reduction in amplitude), while high frequencies would be significantly reduced. A bandstop filter would severely attenuate a middle band of frequencies while passing frequencies above and below the attenuated frequencies. Filter selectivity will be the focus of the first section in this chapter.

Filters can also be described by the method used to approximate the ideal filter. Some approximation methods emphasize low distortion in the passband of the filter while others stress the ability of the filter to attenuate the signals in the stopband. Each approximation method has visible characteristics that distinguish it from the others. Most notably, the absence or presence of ripple (variations) in the passband and stopband clearly set one approximation method apart from another. Filter approximation methods will be discussed in further detail in the second section.

Another means of classifying filters is by the implementation method used. Some filters will be built to filter analog signals using individual components mounted on circuit boards, while other filters might simply be part of a larger digital system which has other functions as well. Several implementation methods will be described in the third section of this chapter as well as the differences between analog and digital signals. However, it should be noted that digital filter design and implementation will be considered in detail starting in Chapter 5, while the first four chapters concentrate on filter approximation theory and analog filter implementation.

In the final section of this chapter we discuss WFilter, an analog and digital filter design package for Windows®, which is included on the software disk. WFilter determines the transfer function coefficients necessary for analog filters or for digital FIR or IIR filters. After the filter has been designed, the user can view the pole-zero plot, as well as the magnitude and phase responses. The filter design parameters or the frequency response parameters can also be edited for ease of use. In addition, for analog filters, the Spice circuit file can be generated to aid in the analysis of active filters. After digital filters have been designed, they may be used to filter wave files and the results can be played for comparison (a sound card must be present). Further discussion of WFilter and the C code supplied with this text can be found in Appendix B.

1.1 FILTER SELECTIVITY

As indicated earlier, a filter's primary purpose is to differentiate between different bands of frequencies, and therefore frequency selectivity is the most common method of classifying filters. Names such as lowpass, highpass, bandpass, and bandstop are used to categorize filters, but it takes more than a name to completely describe a filter. In most cases a precise set of specifications is required in order to allow the proper design of a filter. There are two primary sets of specifications necessary to completely define a filter's response, and each of these can be provided in different ways.

The frequency specifications used to describe the passband(s) and stopband(s) could be provided in hertz (Hz) or in radians/second (rad/sec). We will use the frequency variable f measured in hertz as filter input and output specifications because it is a slightly more common way of discussing frequency. However, the frequency variable ω measured in radians/second will also be used as WFilter's internal variable of choice as well as for unnormalized frequency responses since most of those calculations will use radians/second.

The other major filter specifications are the gain characteristics of the passband(s) and stopband(s) of the filter response. A filter's gain is simply the ratio of the output signal level to the input signal level. If the filter's gain is greater than 1, then the output signal is larger than the input signal, while if the gain is less than 1, the output is smaller than the input. In most filter applications, the gain response in the stopband is very small. For this reason, the gain is typically converted to decibels (dB) as indicated in (1.1). For example, a filter's passband gain response could be specified as 0.707 or as –3.0103 dB, while the stopband gain might be specified as 0.0001 or –80.0 dB.

$$\text{gain}_{\text{dB}} = 20 \cdot \log(\text{gain}) \tag{1.1}$$

As we can see, the values in decibels are more manageable for very small gains. Some filter designers prefer to use attenuation (or loss) values instead of gain values. Attenuation is simply the inverse of gain. For example, a filter with a gain of 1/2 at a particular frequency would have an attenuation of 2 at that frequency. If we express attenuation in decibels we will find that it is simply the negative of the gain in decibels as indicated in (1.2). Gain values expressed in decibels will be the standard quantities used as filter specifications, although the term attenuation (or loss) will be used occasionally when appropriate.

$$\text{attn}_{\text{dB}} = 20 \cdot \log(\text{gain}^{-1}) = -20 \cdot \log(\text{gain}) = -\text{gain}_{\text{dB}} \tag{1.2}$$

1.1.1 Lowpass Filters

Figure 1.1 shows a typical lowpass filter’s response using frequency and gain specifications necessary for precision filter design. The frequency range of the filter specification has been divided into three areas. The passband extends from zero frequency (dc) to the passband edge frequency f_{pass}, and the stopband extends from the stopband edge frequency f_{stop} to infinity. (We will see later in this text that digital filters have a finite upper frequency limit. We will discuss that issue at the appropriate time.) These two bands are separated by the transition band that extends from f_{pass} to f_{stop}. The filter response within the passband is allowed to vary between 0 dB and the passband gain a_{pass}, while the gain in the stopband can vary between the stopband gain a_{stop} and negative infinity. (The 0 dB gain in the passband relates to a gain of 1.0, while the gain of negative infinity in the stopband relates to a gain of 0.0.) A lowpass filter's selectivity can now be specified with only four parameters: the passband gain a_{pass}, the stopband gain a_{stop}, the passband edge frequency f_{pass}, and the stopband edge frequency f_{stop}.

Lowpass filters are used whenever it is important to limit the high-frequency content of a signal. For example, if an old audiotape has a lot of high-frequency “hiss,” a lowpass filter with a passband edge frequency of 8 kHz could be used to eliminate much of the hiss. Of course, it also eliminates high frequencies that were intended to be reproduced. We should remember that any filter can differentiate only between bands of frequencies, not between information and noise.

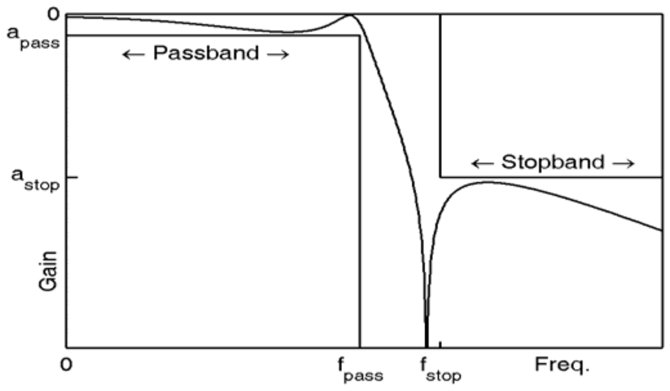


Figure 1.1 Lowpass filter specification.

1.1.2 Highpass Filters

A highpass filter can be specified as shown in Figure 1.2. Note that in this case the passband extends from f_{pass} to infinity (for analog filters) and is located at a higher frequency than the stopband which extends from zero to f_{stop}. The transition band still separates the passband and stopband. The passband gain is still specified as a_{pass} (dB) and the stopband gain is still specified as a_{stop} (dB).

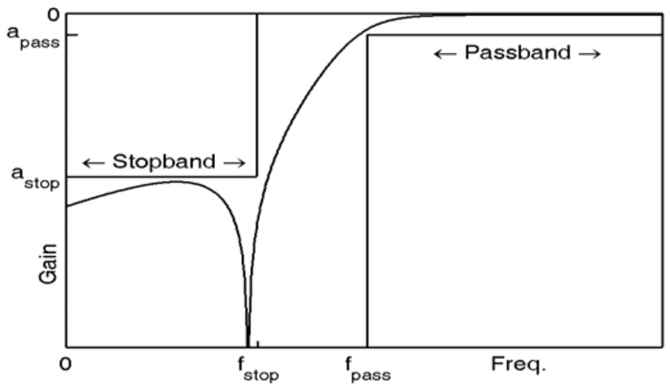


Figure 1.2 Highpass filter specification.

Highpass filters are used when it is important to eliminate low frequencies from a signal. For example, when turntables are used to play LP records (some readers may remember those black vinyl disks that would warp in a car's back window), turntable rumble can sometimes occur, producing distracting low-frequency signals. A highpass filter set to a passband edge frequency of 100 Hz could help to eliminate this distracting signal.

1.1.3 Bandpass Filters

The filter specification for a bandpass filter shown in Figure 1.3 requires a bit more description. A bandpass filter will pass a band of frequencies while attenuating frequencies above or below that band. In this case the passband exists between the lower passband edge frequency f_{pass1} and the upper passband edge frequency f_{pass2}. A bandpass filter has two stopbands. The lower stopband extends from zero to f_{stop1}, while the upper stopband extends from f_{stop2} to infinity (for analog filters). Within the passband, there is a single passband gain parameter a_{pass} in decibels. However, individual parameters for the lower stopband gain a_{stop1} (dB) and the upper stopband gain a_{stop2} (dB) could be used if necessary.

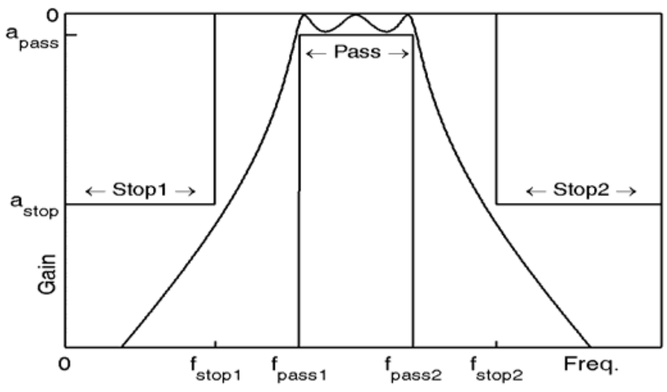


Figure 1.3 Bandpass filter specification.

A good example for the application of a bandpass filter is the processing of voice signals. The normal human voice has a frequency content located primarily in the range of 300–3,000 Hz. Therefore, the frequency response for any system designed to pass primarily voice signals should contain the input signal to that frequency range. In this case, f_{pass1} would be 300 Hz and f_{pass2} would be 3,000 Hz. The stopband edge frequencies would be selected by how fast we would want the signal response to roll off above and below the passband.

1.1.4 Bandstop Filters

The final type of filter to be discussed in this section is the bandstop filter as shown in Figure 1.4. In this case the band of frequencies being rejected is located between the two passbands. The stopband exists between the lower stopband edge frequency f_{stop1} and the upper stopband edge frequency f_{stop2}. The bandstop filter has two passbands. The lower passband extends from zero to f_{pass1}, while the upper passband extends from f_{pass2} to infinity (for analog filters). Within the stopband, the single stopband gain parameter a_{stop} is used. However, individual gain parameters for the lower and upper passbands, a_{pass1} and a_{pass2} (in dB) respectively, could be used if necessary.

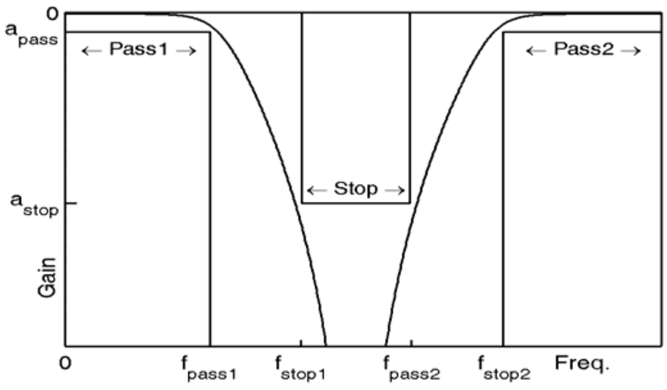


Figure 1.4 Bandstop filter specification.

An excellent example of a bandstop application would be a 60-Hz notch filter used in sensitive measurement equipment. Most electronic measurement equipment today runs from an AC power source using a 60-Hz input frequency. However, it is not uncommon for some of the 60-Hz signal to make its way into the sensitive measurement areas of the equipment. In order to eliminate this troublesome frequency, a bandstop filter (sometimes called a notch filter in these applications) could be used with f_{stop1} set to 58 Hz and f_{stop2} set to 62 Hz. The passband edge frequencies could be adjusted based on the other technical requirements of the filter.

1.2 FILTER APPROXIMATION

The response of an ideal lowpass filter is shown in Figure 1.5, where all frequencies from 0 to f_o are passed with a gain of 1, and all frequencies above f_o are completely attenuated (gain = 0). This type of filter response is physically unattainable. Practical filter responses that can be attained are also shown. As a filter's response becomes closer and closer to the ideal, the cost of the filter (time delay, number of elements, dollars, power consumption, etc.) will increase. These practical responses are referred to as approximations to the ideal. There are a variety of ways to approximate an ideal response based on different criteria. For example, some designs may emphasize the need for minimum distortion of the signals in the passband and would be willing to trade off stopband attenuation for that feature. Other designs may need the fastest transition from passband to stopband and will allow more distortion in the passband to accomplish that aim. It is this engineering tradeoff that makes the design of filters so interesting.

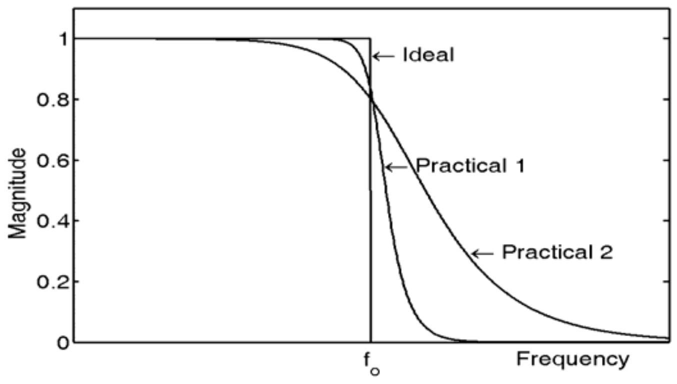


Figure 1.5 Practical and ideal filter responses.

We will be discussing the primary approximation functions used in filter design today that can be classified both by name and the presence of ripple or variation in the signal bands. Elliptic or Cauer filter approximations provide the fastest transition between passband and stopband of any studied in this text. An illustration of the magnitude response of an elliptic filter is shown in Figure 1.1, where we can see that ripple exists in both the passband and stopband. What is not shown in that figure is the phase distortion that the elliptic filter generates. If the filter is to be used with audio signals, this phase distortion must usually be corrected. However, in other applications, for example the transmission of data, the elliptic filter is a popular choice because of its excellent selectivity characteristics. The elliptic approximation is also one of the more complicated to develop. (We will discuss all approximation methods in detail in Chapter 2.)

The inverse Chebyshev response is another popular approximation method that has a smooth response in the passband, but variations in the stopband. On the other hand, the normal Chebyshev response has ripple in the passband, but a smooth, ever-decreasing gain in the stopband. The phase distortion produced by these filters is not as severe as for the elliptic filter, and they are typically easier to design. The inverse Chebyshev response is shown in Figure 1.2, and the normal Chebyshev response is illustrated in Figure 1.3. The Chebyshev approximations provide a good compromise between the elliptic and Butterworth approximation.

The Butterworth filter is a classic filter approximation that has a smooth response in both the passband and stopband as shown in Figure 1.4. It provides the most linear phase response of any approximation technique discussed in this text. (The Bessel approximation provides better phase characteristics, but has very poor transition band characteristics.) However, as we will see in Chapter 2, a

Butterworth filter will require a much higher order to match the transition band characteristics of a Chebyshev or elliptic filter.

1.3 FILTER IMPLEMENTATION

After a filter has been completely specified, various numerical coefficients can be calculated (as described in Chapter 2). But after all the paperwork has been completed, the filter still has to be placed into operation. The first major decision is whether to use analog or digital technology to implement the filter. The differences between analog and digital filter design are based primarily on the differences between analog and digital signals themselves. Any signal can be represented in the time domain by plotting its amplitude versus time. However, the amplitude and time variations can be either continuous or discrete. If both the amplitude and time variations are continuous as shown in Figure 1.6, the signal is referred to as an analog signal. Most real-life signals are analog in nature; for example, sounds that we hear, electrocardiogram signals recorded in a medical lab, and seismic variations recorded on monitoring equipment. However, the problem with analog signals is that they contain so much information. The exact amplitude of the signal (with infinite precision) is available at every instant of time. Do we actually need all of that information? And how do we store and transfer that information?

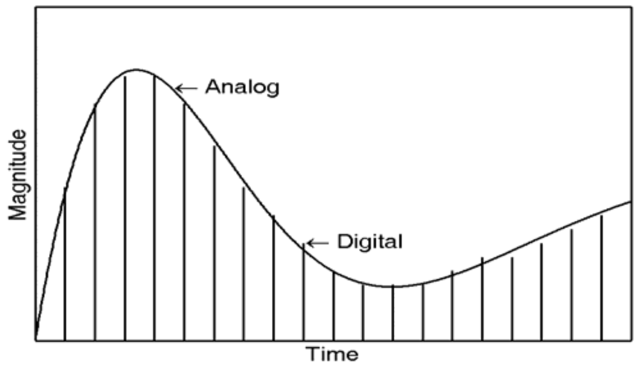


Figure 1.6 Comparison of analog and digital signals.

As techniques for the storage and transmission of information in digital form are becoming more efficient and cost effective, it is increasingly advantageous to use signals that are in a digital form. The advantage of using the resulting digital signals is that the amount of information can be managed to a level appropriate for each application. An analog signal can be converted to a digital signal in two steps. First, the signal must be sampled at fixed time intervals, and then the amplitude of the signal must be quantized to one of a set of fixed levels. Once the analog signal has been converted to a discrete-time and discrete-amplitude signal it is commonly referred to as a digital signal as shown in Figure 1.6. The operation of sampling and quantizing is accomplished by an analog-to-digital converter (ADC). After filtering the signal in the digital domain, a digital-to-analog converter (DAC) can be used to return the signal to analog form. (A more complete discussion of these operations will be given in Chapter 5 where digital filter design is introduced and in Chapter 8 where practical considerations of digital filter implementation are discussed.) Today the digital images and sound files on computers as well as the music on compact discs are examples of signals in digital form.

If we choose to implement a filter in analog form, we still have further choices to make. We could choose to implement the filter with purely passive components such as resistors, capacitors, and inductors. This approach might be the best choice when high frequencies or high power is used. In other circumstances, analog active filters might be the best choice where either transistors or operational amplifiers are used to provide a gain element in the filter. The implementation of analog active filters is considered in Chapter 4.

Digital filters will be implemented by using digital technology available today. Generally, the process will take place within a microprocessor system that could have other functions besides the filtering of signals. We discuss two basic types of digital filters in Chapters 6 and 7 of this text. The first is the infinite impulse response (IIR) digital filter that is based in a large part on the design methodology of analog filters. The second type is the finite impulse response (FIR) digital filter that uses a completely different method for its design. The implementation of digital filters is considered in Chapter 8. Chapter 9 introduces the fast Fourier transform (FFT) and discusses how it can be used in filtering.

As we can now see, there is more to describing a filter than referring to it as a lowpass filter. For example, we might be designing an “analog active lowpass Butterworth filter” or a “digital IIR bandpass Chebyshev filter.” These names along with a filter's specification parameters will completely describe a filter.

1.4 WFILTER - FILTER DESIGN SOFTWARE

Although we haven’t discussed filter design in detail, it may be educational to see how to use a filter design software package. This book includes a filter design software package called WFilter that automates the design process and provides filter coefficients and frequency response characteristics for the filters, and other features as well. As a first example, we will choose an analog lowpass Chebyshev filter with passband and stopband gains of -1 dB and -30 dB, respectively. The passband and stopband edge frequencies will be 500 Hz and 1,000 Hz and the data files and frequency plots will be labeled with the title “Lowpass Chebyshev Filter.” (You will need to install WFilter on your computer before you can duplicate the actions described below. Please see Appendix B for contents of the accompanying disc and installation instructions for WFilter.)

After starting WFilter, you can begin the design of a new filter by selecting ***New*** from the ***File*** menu bar as shown in Figure 1.7. You can also ***Open*** a previously designed filter or seek ***Help*** from this startup screen.

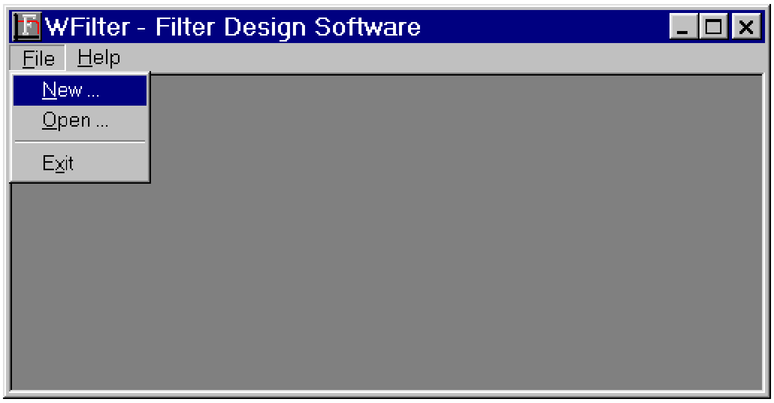
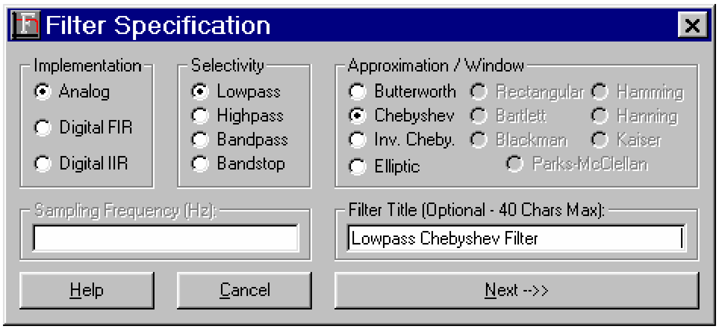


Figure 1.7 WFilter opening screen.

After selecting ***New***, you will be able to select the type of filter you want to design and specify a description as indicated earlier. The ***Filter Specification*** dialog box shown in Figure 1.8 includes sections for the different characteristics of the filter, as well as sampling frequency (for digital filters only) and a filter title.



Implementation: [✓ Analog] Digital FIR Digital IIR

Selectivity: [✓ Lowpass] Highpass Bandpass Bandstop

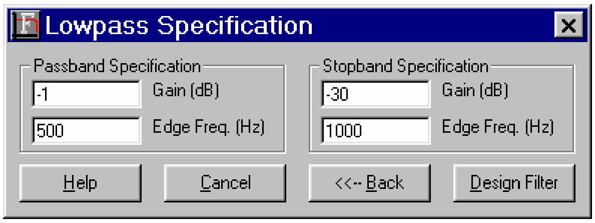
Approximation / Window: Butterworth [✓ Chebyshev] Inv. Cheby. Elliptic Rectangular Bartlett Blackman Hamming Hanning Kaiser Parks-McClellan

Sampling Frequency (Hz):

Filter Title (Optional - 40 Chars Max): Lowpass Chebyshev Filter

Figure 1.8 Filter Specification dialog box.

After selecting the characteristics of the filter, you can select ***Next*** and the ***Lowpass Specification*** dialog box shown in Figure 1.9 will appear, allowing you to specify the gain and frequency characteristics. For a lowpass filter you must enter the passband gain and edge frequency as well as the stopband gain and edge frequency as specified in our sample filter. You then have the option of designing the filter or returning to the previous dialog box to make changes. At each stage of this process you can cancel your actions or seek help determining proper actions.


Lowpass Specification

Passband Specification
-1 Gain (dB)
500 Edge Freq. (Hz)

Stopband Specification
-30 Gain (dB)
1000 Edge Freq. (Hz)

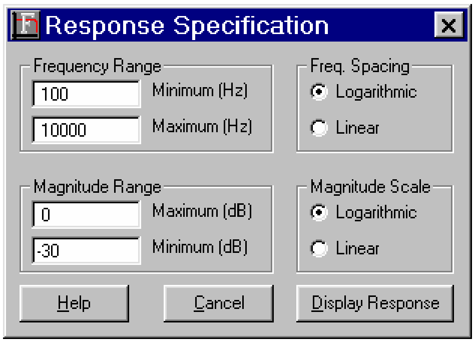
Figure 1.9 Lowpass Specification dialog box.

After completing the specifications, you can select ***Design Filter*** and the following information will be displayed in the ***Filter Parameter*** text window as shown in Figure 1.10. (Pole-zero information will also be displayed, but is not shown in this example.) As indicated, our filter will be fourth-order (length is a term used with digital FIR filters) and will have an overall gain as indicated. The coefficients of the transfer function are given in the form of quadratics. Details concerning these values will be discussed in the next chapter.

```
Lowpass Chebyshev Filter

Selectivity:         Lowpass
Approximation:       Chebyshev
Implementation:      Analog
Passband gain (dB):  -1.0
Stopband gain (dB):  -30.0
Passband freq (Hz):  500.0
Stopband freq (Hz):  1000.0

Filter Length/Order: 04
Overall Filter Gain: 8.91250938134E-01

Numerator Coefficients
QD [S^2 +       S         +        1       ]
== ============================================
01 0.0   0.00000000000E+00   9.73641285912E+06
02 0.0   0.00000000000E+00   2.75754865948E+06

Denominator Coefficients
QD [S^2 +       S         +        1       ]
== ============================================
01 1.0   8.76730519296E+02   9.73641285912E+06
02 1.0   2.11661471023E+03   2.75754865948E+06
```

Figure 1.10 Filter Characteristic screen.

We can now display the frequency response of our filter by selecting ***Magnitude Response*** from the ***View*** menu. As indicated in the ***Response Specification*** dialog box shown in Figure 1.11, the user can specify frequency limits of the display as well as the magnitude range. Both the frequency and magnitude axes can be scaled in a linear or logarithmic fashion. WFilter initializes the values with educated guesses that can be changed by the user.


Response Specification

Frequency Range
100 Minimum (Hz)
10000 Maximum (Hz)

Freq. Spacing
(●) Logarithmic
( ) Linear

Magnitude Range
0 Maximum (dB)
-30 Minimum (dB)

Magnitude Scale
(●) Logarithmic
( ) Linear

Figure 1.11 Response Specification dialog box.

Selecting ***Display Response*** will bring up the graphics plots of the magnitude and phase responses. These graphs are shown in Figures 1.12 and 1.13. Notice that on the phase plot, the phase angle is always displayed as a value between +180 and –180 degrees. Therefore there is actually no discontinuity in the phase plot.

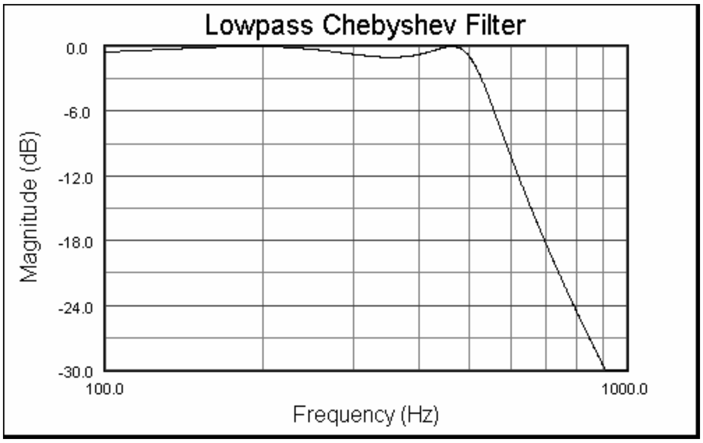


Figure 1.12 Magnitude response screen.

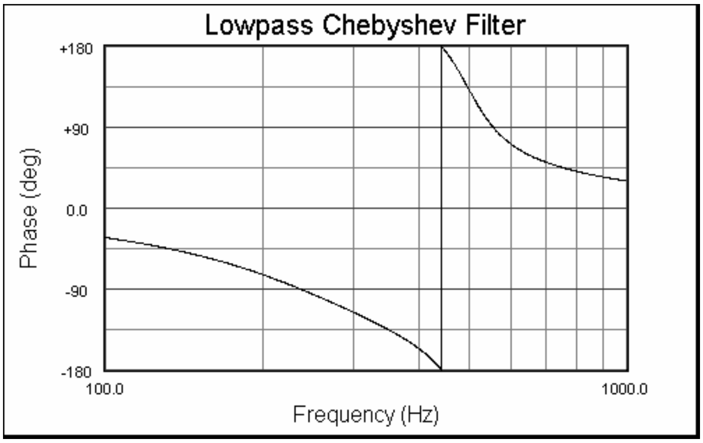


Figure 1.13 Phase response screen.

In addition, for analog and digital IIR filters, the pole-zero positions for a filter can be displayed by selecting ***Pole-Zero Plot*** from the ***View*** menu. (The significance of poles and zeros will be discussed in the next chapter.) The pole-zero plot for our filter is shown in Figure 1.14.

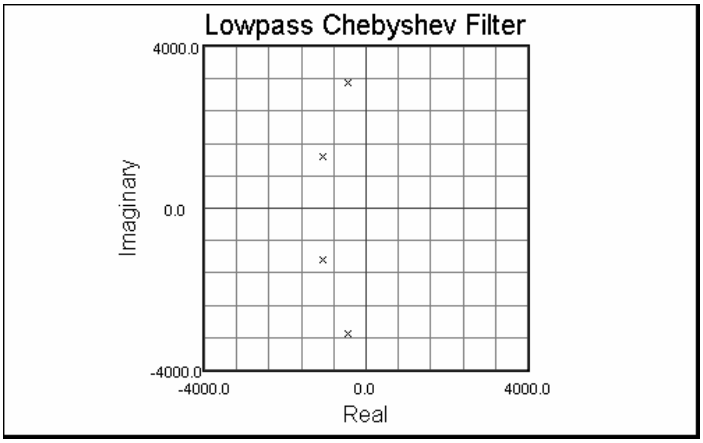
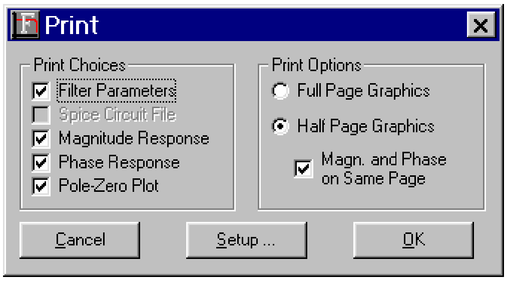


Figure 1.14 Pole-zero plot.

Any or all of the information provided by WFilter can be printed by selecting ***Print*** from the ***File*** menu. The Print dialog box as shown in Figure 1.15 will be displayed and the user can choose from several options for printing.

The details of this filter design can now be saved by selecting ***Save*** or ***Save As*** from the ***File*** menu. After saving the information, you can exit the program by selecting ***Exit*** from the ***File*** menu. Further details about the WFilter program will be presented in the chapters to come. We will have many more chances to use this program as we develop the theory behind analog and digital filter design.



Print

Print Choices
- [✓] Filter Parameters
- [ ] Spice Circuit File
- [✓] Magnitude Response
- [✓] Phase Response
- [✓] Pole-Zero Plot

Print Options
- Full Page Graphics
- [✓ Half Page Graphics]
  - [✓] Magn. and Phase on Same Page

Figure 1.15 Print dialog box.

1.5 CONCLUSION

At this point we have provided an introduction to filter design by providing the standard definitions used to describe filters. We have also introduced WFilter, a powerful filter design software package. But we still need to learn the theory behind filter design and how we can design and implement the filters. We'll begin in the next chapter to study the design of analog filters. For those interested in the C code used in the design and implementation of filters, please refer to Appendix C–I.

Chapter 2

Analog Filter Approximation Functions

As indicated in the first chapter, an ideal filter is unattainable; the best we can do is to approximate it. There are a number of approximations we can use based on how we want to define "best." In this chapter we discuss four methods of approximation, each using a slightly different definition. Four sections are devoted to the major approximation methods used in analog filter design: the Butterworth, Chebyshev, inverse Chebyshev, and elliptic approximations. In each of these sections we determine the order of the filter required given the filter's specifications and the required normalized transfer function to satisfy the specifications. In the following section we discuss the relative advantages and disadvantages of using these approximation methods. But first we begin this chapter by describing analog filters mathematically in the form of linear system transfer functions.

2.1 FILTER TRANSFER FUNCTIONS

An analog filter is a linear system that has an input and output signal. This system's primary purpose is to change the frequency response characteristics of the input signal as it moves through the filter. The characteristics of this filter system could be studied in the time domain or the frequency domain. From a systems point of view, the impulse response $h(t)$ could be used to describe the system in the time domain. The impulse response of a system is the output of a system that has had an impulse applied to the input. Of course, many systems would not be able to sustain an infinite spike (the impulse) being applied to the input of the system, but there are ways to determine $h(t)$ without actually applying the impulse.

A filter system can also be described in the frequency domain by using the transfer function $H(s)$. The transfer function of the system can be determined by finding the Laplace transform of $h(t)$. Figure 2.1 indicates that the filter system can be considered either in the time domain or in the frequency domain. However, the transfer function description is the predominant method used in filter design, and we will perform most of our filter design using it.

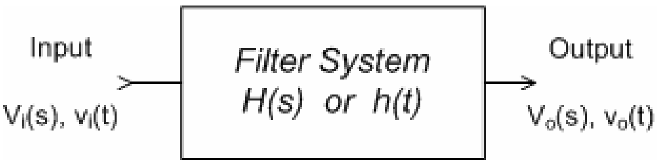


Figure 2.1 The filter as a system.

2.1.1 Transfer Function Characterization

The transfer function $H(s)$ for a filter system can be characterized in a number of ways. As shown in (2.1), $H(s)$ is typically represented as the ratio of two polynomials in s where in this case the numerator polynomial is order m and the denominator is a polynomial of order n. G represents an overall gain constant that can take on any value.

$$H(s) = \frac{G \cdot [s^m + a_{m-1} \cdot s^{m-1} + a_{m-2} \cdot s^{m-2} + \cdots + a_1 \cdot s + a_0]}{[s^n + b_{n-1} \cdot s^{n-1} + b_{n-2} \cdot s^{n-2} + \cdots + b_1 \cdot s + b_0]} \tag{2.1}$$

Alternately, the polynomials can be factored to give a form as shown in (2.2). In this representation, the numerator and denominator polynomials have been separated into first-order factors. The zs represent the roots of the numerator and are referred to as the zeros of the transfer function. Similarly, the ps represent the roots of the denominator and are referred to as the poles of the transfer function.

$$H(s) = \frac{G \cdot [(s+z_0) \cdot (s+z_1) \cdots (s+z_{m-2}) \cdot (s+z_{m-1})]}{[(s+p_0) \cdot (s+p_1) \cdots (s+p_{n-2}) \cdot (s+p_{n-1})]} \tag{2.2}$$

Most of the poles and zeros in filter design will be complex valued and will occur as complex conjugate pairs. In this case, it will be more convenient to represent the transfer function as a ratio of quadratic terms that combine the individual complex conjugate factors as shown in (2.3). The first-order factors that are included will be present only if the numerator or denominator polynomial orders are odd. We will be using this form for most of the analog filter design material.

$$H(s) = \frac{G \cdot [(s+z_0) \cdot (s^2 + a_{01} \cdot s + a_{02}) \cdots (s^2 + a_{q1} \cdot s + a_{q2})]}{[(s+p_0) \cdot (s^2 + b_{01} \cdot s + b_{02}) \cdots (s^2 + b_{r1} \cdot s + b_{r2})]} \tag{2.3}$$

As an example of each expression, consider the three forms of a transfer function that have a second-order numerator and third-order denominator:

$$H_a(s) = \frac{6.0 \cdot (s^2 + 0.66667)}{s^3 + 3.1650 \cdot s^2 + 5.0081 \cdot s + 4.0001} \tag{2.4a}$$

$$H_b(s) = \frac{6.0 \cdot (s + j0.81650)(s - j0.81650)}{(s + 1.5975) \cdot (s + 0.7837 + j1.3747) \cdot (s + 0.7837 - j1.3747)} \tag{2.4b}$$

$$H_c(s) = \frac{6.0 \cdot (s^2 + 0.66667)}{(s + 1.5975) \cdot (s^2 + 1.5675 \cdot s + 2.5040)} \tag{2.4c}$$

2.1.2 Pole-Zero Plots and Transfer Functions

When the quadratic form of the transfer function is used, it is easy to generate the pole-zero plot for a particular transfer function. The pole-zero plot simply plots the roots of the numerator (zeros) and the denominator (poles) on the complex *s*-plane. As an example, the pole-zero plot for the sample transfer function given in (2.4) is shown in Figure 2.2.

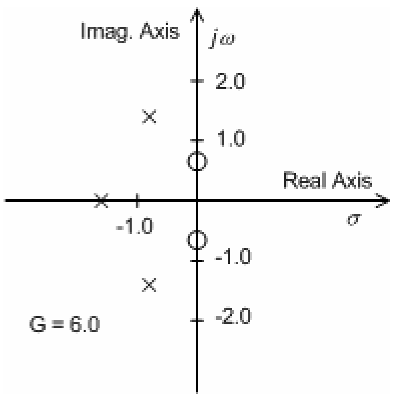


Figure 2.2 Pole-zero plot for (2.4).

A pole is traditionally represented by an *X* and a zero by an *O*. If the transfer function is odd, the first-order pole or zero will be located on the real axis. All poles and zeros from the quadratic factors are symmetrically located pairs in the complex plane on opposite sides of the real axis. The gain of the transfer function must be indicated on the plot or the information would be incomplete. Note that there are only two zeros shown, but there is one located at infinity. We can verify this by observing that if we were to allow $|s|$ to approach infinity, $|H(s)|$ would approach zero. Transfer functions always have the same number of poles and zeros, but some exist at infinity.

Conversely, we can also determine a filter's transfer function from the pole-zero plot. In general, any critical frequency (pole or zero) is specified by indicating the real (σ) and imaginary (ω) component. The transfer function would then include a factor of $[s - (\sigma + j\omega)]$. If the critical frequency is complex, we can combine the two complex conjugate factors into a single quadratic factor by multiplying them as shown in (2.5):

$$[s-(\sigma+j\omega)]\cdot[s-(\sigma-j\omega)] = s^2 - 2\cdot\sigma\cdot s + (\sigma^2+\omega^2) \tag{2.5}$$

<u>Example 2.1 Generating a Transfer Function from a Pole-Zero Plot</u>

Problem: Assume that a pole-zero plot shows poles at ($-3 \pm j2$) and (-4.5) and zeros at ($-5 \pm j1$) and (-1). Determine the transfer function if its gain is 1.0 at $s = 0$.

Solution: Using the technique of (2.5), the complex conjugate poles and zeros can be combined into quadratic factors as indicated. The first-order factors are handled directly and the gain is included in the numerator. The easiest method to use when given a gain requirement of 1.0 at $s = 0$ is to prepare each factor independently to have a gain of 1 at that frequency as shown in the first transfer function. Then the set of constants can be combined as shown in the second equation.

$$H(s) = \frac{(s^2+10\cdot s+26)}{26}\cdot\frac{(s+1)}{1}\cdot\frac{4.5}{(s+4.5)}\cdot\frac{13}{(s^2+6\cdot s+13)}$$

$$= \frac{2.25\cdot(s+1)\cdot(s^2+10\cdot s+26)}{(s+4.5)\cdot(s^2+6\cdot s+13)}$$

2.1.3 Normalized Transfer Functions

In this chapter we concentrate on developing what is referred to as a normalized transfer function. A normalized lowpass transfer function is one in which the passband edge radian frequency is set to 1 rad/sec. Of course, this seems a rather unusual frequency, since seldom would a lowpass filter be required to have such a low frequency. However, the technique actually allows the filter designer considerable latitude in designing filters because a normalized transfer function can easily be unnormalized to any other frequency. In the next chapter, we will discuss in detail the procedures used to unnormalize lowpass filters to other frequencies and even learn how to translate a lowpass filter to a highpass, bandpass, or bandstop filter.

Before we begin the development of the approximation functions for analog filters, it may be helpful to go over the general approach taken in these sections. In each case, the general characteristics of the approximation method will be discussed, including its relative advantages and disadvantages. Next, a description of the transfer function for each approximation will be given. There will be no attempt to give an exhaustive derivation of each approximation method in this text; there are more than enough sources of theoretical developments already available. (A list of references for material presented in this chapter is given in Appendix A. The texts by Daniels, Van Valkenburg, and Parks/Burrus are particularly helpful when studying approximation theory.) We will then determine numerical methods to find the order and the coefficients of the transfer function necessary to meet the filter specifications.

2.2 BUTTERWORTH NORMALIZED APPROXIMATION FUNCTIONS

The Butterworth approximation function is often called the maximally flat response because no other approximation has a smoother transition through the passband to the stopband. The phase response also is very smooth, which is important when considering distortion. The lowpass Butterworth polynomial has an all-pole transfer function with no finite zeros present. It is the approximation method of choice when low phase distortion and moderate selectivity are required.

2.2.1 Butterworth Magnitude Response

Equation 2.6 gives the Butterworth approximation's magnitude response where ω_o is the passband edge frequency for the filter, *n* is the order of the approximation function, and ε is the passband gain adjustment factor. The transfer functions will carry subscripts to help identify them in this chapter. In this case, the subscript *B* indicates a Butterworth filter, and *n* indicates an *n*th-order transfer function.

$$\left| H_{B,n}\left[j(\omega/\omega_o)\right]\right| = \frac{1}{\sqrt{1+\varepsilon^2 \cdot (\omega/\omega_o)^{2\cdot n}}} \tag{2.6}$$

where

$$\varepsilon = \sqrt{10^{-0.1\cdot a_{\text{pass}}} - 1} \tag{2.7}$$

If we set both ε = 1 and $\omega_o = 1$, the filter will have a gain of 1/2 or –3.01 dB at the normalized passband edge frequency of 1 rad/sec.

The Butterworth approximation has a number of interesting properties. First, the response will always have unity gain at ω = 0, no matter what value is given to ε. However, the gain at the normalized passband edge frequency of ω = 1 will depend on the value of ε. In addition, the response gain decreases by a factor of $-20n$ dB per decade of frequency change. That happens because for large ω, the transfer function gain becomes inversely proportional to ω, which increases by 10 for every decade. (A decade in frequency is a ratio of 10. For example, the span of frequencies from 1 to 10 rad/sec and the span of frequencies from 1,000 to 10,000 Hz are both referred to as one decade.) Therefore, if we design a fifth-order Butterworth filter, the gain will decrease 100 dB per decade for frequencies above the passband edge frequency.

2.2.2 Butterworth Order

The order of the Butterworth filter is dependent on the specifications provided by the user. These specifications include the edge frequencies and gains. The standard formula for the Butterworth order calculation is given in (2.8). In this formulation, note that it is the ratio of the stopband and passband frequencies which is important, not either one of these independently. This means that a filter with a given set of gains will require the same order whether the edge frequencies are 100 and 200 rad/sec or 100,000 and 200,000 Hz. The value of *n* calculated using this equation must always be rounded to the next highest integer in order to guarantee that the specifications will be met by the integer order of the filter designed:

$$n_B = \frac{\log[(10^{-0.1 \cdot a_{\text{stop}}} - 1)/(10^{-0.1 \cdot a_{\text{pass}}} - 1)]}{2 \cdot \log(\omega_{\text{stop}} / \omega_{\text{pass}})} \tag{2.8}$$

2.2.3 Butterworth Pole Locations

The poles for a Butterworth approximation function are equally spaced around a circle in the *s*-plane and are symmetrical about the *j*ω axis. Plotting the poles of the magnitude-squared function $|H(s)|^2$ shows twice as many poles as the order of the filter. We are able to determine the Butterworth transfer function from the poles in the left half plane (LHP) that produce a stable system. In order to determine the exact pole positions in the *s*-plane we use the polar form for specifying the complex location. For each of the poles, we must know the distance from the origin (the radius of the circle) and the angle from the positive real axis.

The radius of the circle for our normalized case is a function of the passband gain and is given in (2.9):

$$R = \varepsilon^{-1/n} \tag{2.9}$$

Once the radius of the circle is known, the pole positions are determined by calculating the necessary angles. Equation 2.10 can be used to determine the angles for those complex poles in the second quadrant:

$$\theta_m = \frac{\pi \cdot (2 \cdot m + n + 1)}{2 \cdot n}, m = 0, 1, \ldots, (n/2) - 1 \ (n \text{ even}) \tag{2.10a}$$

$$\theta_m = \frac{\pi \cdot (2 \cdot m + n + 1)}{2 \cdot n}, m = 0, 1, \ldots, [(n-1)/2] - 1 \ (n \text{ odd}) \tag{2.10b}$$

It is important to remember that in this equation θ_m represents only the angles in the second quadrant *that have complex conjugates in the third quadrant*. In other words, θ_m does *not* include the pole on the real axis for odd-order functions. For this reason, (2.10b) is valid only for odd-order filters where $n \geq 3$ since a first-order filter would have no complex conjugate poles. (We'll see that this definition allows a cleaner algorithm for the C code that is discussed in Appendix D.) The precise pole locations can then be determined from (2.11) and (2.12):

$$\sigma_m = R \cdot \cos(\theta_m) \tag{2.11}$$

$$\omega_m = R \cdot \sin(\theta_m) \tag{2.12}$$

In the case of odd-order transfer functions, the first-order pole will be located at a position σ_R equal to the radius of the circle as indicated in (2.13):

$$\sigma_R = -R \tag{2.13}$$

2.2.4 Butterworth Transfer Functions

The Butterworth transfer function can be determined from the pole locations in the LHP as we saw in the first section of this chapter. Since most of these poles are complex conjugate pairs (except for the possible pole on the real axis for odd orders), we can get all of the information we need from the poles in the second quadrant. The complete approximation transfer function can be determined from a combination of a first-order factor (for odd orders) and quadratic factors. Each of these factors will have a constant in the numerator to adjust the gain to unity at ω = 0 as illustrated in Example 2.1. We start by defining the form of the first-order factor in (2.14). Note that at this point the transfer function variables are represented by an uppercase *S* where prior to this we have been using a lowercase *s*. This is an attempt to distinguish between the normalized transfer function (using *S*) and the unnormalized functions (using *s*), which will be developed in the next chapter.

$$H_o(S) = \frac{R}{S+R} \tag{2.14}$$

For each complex conjugate pole in the second quadrant, there will be the following quadratic factor in the transfer function:

$$H_m(S) = \frac{B_{2m}}{S^2 + B_{1m} \cdot S + B_{2m}} \tag{2.15}$$

where

$$B_{1m} = -2 \cdot \sigma_m \tag{2.16}$$

$$B_{2m} = \sigma_m^2 + \omega_m^2 \tag{2.17}$$

The complete Butterworth transfer function can now be defined as shown in (2.18):

$$H_{B,n}(S) = \frac{\prod_m (B_{2m})}{\prod_m (S^2 + B_{1m} \cdot S + B_{2m})}, \quad m = 0,1,\ldots,(n/2)-1 \ (n \text{ even}) \tag{2.18a}$$

$$H_{B,n}(S) = \frac{R \cdot \prod_m (B_{2m})}{(S+R) \cdot \prod_m (S^2 + B_{1m} \cdot S + B_{2m})}, \quad m = 0,1,\ldots,[(n-1)/2]-1 \ (n \text{ odd}) \tag{2.18b}$$

We have now reached a point where some examples are in order. First, we will consider some numerical examples, and then test the WFilter program on the same specifications.

Example 2.2 Butterworth Third-Order Normalized Transfer Function

Problem: Determine the order, pole locations, and transfer function coefficients for a Butterworth filter to satisfy the following specifications:

$$a_{\text{pass}} = -1 \text{ dB},\ a_{\text{stop}} = -12 \text{ dB},\ \omega_{\text{pass}} = 1 \text{ rad/sec, and } \omega_{\text{stop}} = 2 \text{ rad/sec}$$

Solution: First, we determine the fundamental constants needed from (2.7)–(2.9):

$\varepsilon = 0.508847$ $n = 2.92$ (3rd order) $R = 1.252576$

Next, we find the locations of the first-order pole and the complex pole in the second quadrant from (2.10)–(2.13). A graph of the pole locations (including those from the magnitude-squared function in the right-half plane) is shown in Figure 2.3.

(1st order)	$\sigma_R = -1.252576$	$\omega_R = 0.0$
$\theta_0 = 2\pi/3$	$\sigma_0 = -0.626288$	$\omega_0 = 1.084763$

Finally, we generate the transfer function from (2.14)–(2.18):

$$H_{B,3}(S) = \frac{1.2526 \cdot 1.5689}{(S + 1.2526) \cdot (S^2 + 1.2526 \cdot S + 1.5689)}$$

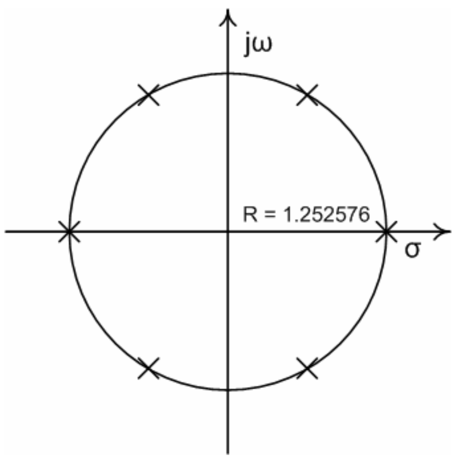


Figure 2.3 Pole locations for third-order Butterworth normalized filter.

In order to use WFilter to determine the normalized transfer functions, we will assume a passband edge frequency of 1 rad/sec (0.159154943092 Hz) and a stopband edge frequency of 2 rad/sec (0.318309886184 Hz). (We must enter the frequencies into WFilter using the hertz values and they must have more significant digits than we require in our answer.) Twelve significant digits were used to enter the edge frequencies, but they are displayed on the coefficient screen with only ten significant digits. Rest assured that they are stored internally with the higher accuracy, but the display is set for the more typical requirements of filter frequency. The coefficient values determined are shown in Figure 2.4.

```
Butterworth 3rd-Order Normalized Lowpass

Selectivity:        Lowpass
Approximation:      Butterworth
Implementation:     Analog
Passband gain (dB): -1.0
Stopband gain (dB): -12.0
Passband freq (Hz): 0.1591549431
Stopband freq (Hz): 0.3183098862

Filter Length/Order: 03
Overall Filter Gain: 1.00000000000E+00

            Numerator Coefficients
QD [S^2 +       S         +        1       ]
== ============================================
01 0.0  0.00000000000E+00  1.25257638818E+00
02 0.0  0.00000000000E+00  1.56894760823E+00

           Denominator Coefficients
QD [S^2 +       S         +        1       ]
== ============================================
01 0.0  1.00000000000E+00  1.25257638818E+00
02 1.0  1.25257638818E+00  1.56894760823E+00
```

Figure 2.4 Butterworth normalized third-order coefficients from WFilter.

Example 2.3 Butterworth Fourth-Order Normalized Transfer Function

Problem: Determine the order, pole locations, and transfer function coefficients for a Butterworth filter to satisfy the following specifications:

$$a_{\text{pass}} = -1 \text{ dB}, \ a_{\text{stop}} = -18 \text{ dB}, \ \omega_{\text{pass}} = 1 \text{ rad/sec, and } \omega_{\text{stop}} = 2 \text{ rad/sec}$$

Solution: First, we determine the constants needed from (2.7)–(2.9):

$\varepsilon = 0.508847$ $\qquad$ $n = 3.95$ (4th order) $\qquad$ $R = 1.184004$

Next, we find the locations of the two complex poles in the second quadrant from (2.10)–(2.13). A graph of the pole locations is shown in Figure 2.5.

$\theta_0 = 5\pi/8$ $\qquad$ $\sigma_0 = -0.453099$ $\qquad$ $\omega_0 = +1.093877$

$\theta_1 = 7\pi/8$ $\qquad$ $\sigma_1 = -1.093877$ $\qquad$ $\omega_1 = +0.453099$

Finally, we generate the transfer function from (2.14)–(2.18). The coefficient values determined by WFilter are shown in Figure 2.6.

$$H_{B,4}(S) = \frac{(1.4019)^2}{(S^2 + 2.1878 \cdot S + 1.4019) \cdot (S^2 + 0.90620 \cdot S + 1.4019)}$$

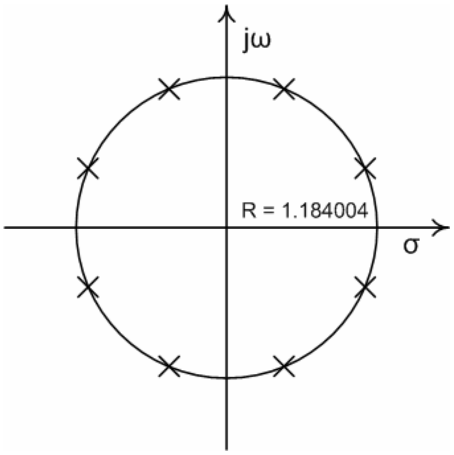


Figure 2.5 Pole locations for fourth-order Butterworth normalized filter.

```
Butterworth 4th-Order Normalized Lowpass

Selectivity:         Lowpass
Approximation:       Butterworth
Implementation:      Analog
Passband gain (dB): -1.0
Stopband gain (dB): -18.0
Passband freq (Hz): 0.1591549431
Stopband freq (Hz): 0.3183098862

Filter Length/Order: 04
Overall Filter Gain: 1.00000000000E+00

            Numerator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 0.0  0.00000000000E+00  1.40186544588E+00
02 0.0  0.00000000000E+00  1.40186544588E+00

           Denominator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 1.0  9.06197420862E-01  1.40186544588E+00
02 1.0  2.18775410363E+00  1.40186544588E+00
```

Figure 2.6 Butterworth normalized fourth-order coefficients from WFilter.

The associated magnitude and phase responses for the previous two examples are shown in Figures 2.7 and 2.8 and illustrate the difference between a third-order and fourth-order filter. Notice that the magnitude response uses a different scale for the passband and stopband response. (WFilter does not put two different responses on the same graph or use different scales for passband and stopband. They are displayed here in that manner for ease of comparison. However, the magnitude scale of WFilter can be changed on different graphs to provide more detail.)

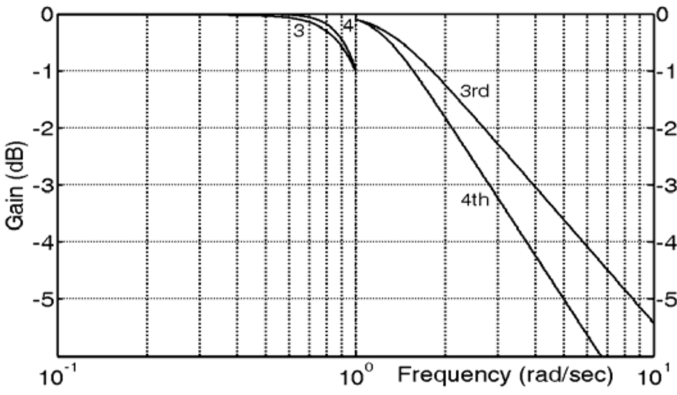


Figure 2.7 Butterworth third-order and fourth-order magnitude responses.

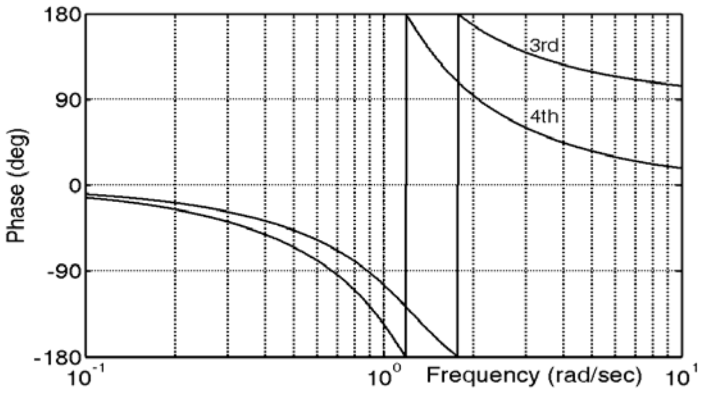


Figure 2.8 Butterworth third-order and fourth-order phase responses.

2.3 CHEBYSHEV NORMALIZED APPROXIMATION FUNCTIONS

The Chebyshev approximation function also has an all-pole transfer function like the Butterworth approximation. However, unlike the Butterworth case, the Chebyshev filter allows variation or ripple in the passband of the filter. This reduction in the restrictions placed on the characteristics of the passband enables the transition characteristics of the Chebyshev to be steeper than the Butterworth transition. Because of this more rapid transition, the Chebyshev filter is able to satisfy user specifications with lower-order filters than the Butterworth case. However, the phase response is not as linear as the Butterworth case, and therefore if low phase distortion is a priority, the Chebyshev approximation may not be the best choice.

2.3.1 Chebyshev Magnitude Response

The magnitude response function for the Chebyshev approximation is shown in (2.19):

$$\left|H_{C,n}[j(\omega/\omega_o)]\right| = \frac{1}{\sqrt{1+\varepsilon^2 \cdot C_n^2(\omega/\omega_o)}} \tag{2.19}$$

where the definition of ε is again

$$\varepsilon = \sqrt{10^{-0.1\cdot a_{\text{pass}}} - 1} \tag{2.20}$$

and $C_n(\omega)$ is the Chebyshev polynomial of the first kind of degree *n*. The normalized Chebyshev polynomial ($\omega_o = 1$) is defined as

$$C_n(\omega) = \cos[n \cdot \cos^{-1}(\omega)],\ \omega \le 0 \tag{2.21a}$$

$$C_n(\omega) = \cosh[n \cdot \cosh^{-1}(\omega)],\ \omega > 0 \tag{2.21b}$$

We can see that the mathematical description used for this approximation is more involved than the Butterworth case. We will be concerned with the expression where ω > 0, but the Chebyshev polynomial has many interesting features which are discussed in the references at the end of this text.

2.3.2 Chebyshev Order

The order of the Chebyshev filter will be dependent on the specifications provided by the user. The general form of the calculation for the order is the same as for the Butterworth, except that the inverse hyperbolic cosine function is used in place of the common logarithm function. As in the Butterworth case, the value of *n* actually calculated must be rounded to the next highest integer in order to guarantee that the specifications will be met.

$$n_C = \frac{\cosh^{-1}\left[\sqrt{(10^{-0.1 \cdot a_{\text{stop}}} - 1)/(10^{-0.1 \cdot a_{\text{pass}}} - 1)}\right]}{\cosh^{-1}(\omega_{\text{stop}} / \omega_{\text{pass}})} \tag{2.22}$$

2.3.3 Chebyshev Pole Locations

The poles for a Chebyshev approximation function are located on an ellipse instead of a circle as in the Butterworth case. The ellipse is centered at the origin of the *s*-plane with its major axis along the jω axis with intercepts of ± cosh(*D*), while the minor axis is along the real axis with intercepts of ± sinh(*D*). The variable *D* is defined as

$$D = \frac{\sinh^{-1}(\varepsilon^{-1})}{n} \tag{2.23}$$

The pole locations can be defined in terms of *D* and an angle ϕ as shown in (2.24). The angles determined locate the poles of the transfer function in the first quadrant. However, we can use them to find the poles in the second quadrant by simply changing the sign of the real part of each complex pole. The real and imaginary components of the pole locations can now be defined as shown in (2.25) and (2.26):

$$\phi_m = \frac{\pi \cdot (2 \cdot m + 1)}{2 \cdot n}, m = 0, 1, \ldots, (n/2) - 1 \ (n \text{ even}) \tag{2.24a}$$

$$\phi_m = \frac{\pi \cdot (2 \cdot m + 1)}{2 \cdot n}, m = 0, 1, \ldots, [(n - 1)/2] - 1 \ (n \text{ odd}) \tag{2.24b}$$

$$\sigma_m = -\sinh(D) \cdot \sin(\phi_m) \tag{2.25}$$

$$\omega_m = \cosh(D) \cdot \cos(\phi_m) \tag{2.26}$$

If the function has an odd-order, there will be a real pole located in the LHP as indicted by (2.27):

$$\sigma_R = -\sinh(D) \tag{2.27}$$

2.3.4 Chebyshev Transfer Functions

Using the results of (2.27), we know that an odd-order Chebyshev transfer function will have a factor of the form illustrated in (2.28):

$$H_o(S) = \frac{\sinh(D)}{S + \sinh(D)} \tag{2.28}$$

The quadratic factors for the Chebyshev transfer function will take on exactly the same form as the Butterworth case, as shown below:

$$H_m(S) = \frac{B_{2m}}{S^2 + B_{1m} \cdot S + B_{2m}} \tag{2.29}$$

$$B_{1m} = -2 \cdot \sigma_m \tag{2.30}$$

$$B_{2m} = \sigma_m^2 + \omega_m^2 \tag{2.31}$$

We are now just about ready to define the general form of the Chebyshev transfer function. However, one small detail still must be considered. Because there is ripple in the passband, Chebyshev even and odd-order approximations do *not* have the same gain at ω = 0. As seen in Figure 2.13 (a result of a future example), each approximation has a number of half-cycles of ripple in the passband equal to the order of the filter. This forces even-order filters to have a gain of a_{pass} at ω = 0. However, the first-order and quadratic factors we have defined are all set to give 0 dB gain at ω = 0. Therefore, if no adjustment of gain is made to even-order Chebyshev approximations, they would have a gain of 0 dB at ω = 0 and a gain of $-a_{\text{pass}}$ (that is, a gain greater than 1.0) at certain other frequencies where the ripple peaks. A gain constant must therefore be included for even-order transfer functions with the value of

$$G = 10^{0.05 \cdot a_{\text{pass}}} \tag{2.32}$$

We are now ready to define a generalized transfer function for the Chebyshev approximation function as shown below:

$$H_{C,n}(S) = \frac{(10^{0.05 \cdot a_{\text{pass}}}) \cdot \prod_m (B_{2m})}{\prod_m (S^2 + B_{1m} \cdot S + B_{2m})}, \tag{2.33a}$$

$$m = 0, 1, \ldots, (n/2) - 1 \ \ (n \text{ even})$$

$$H_{C,n}(S) = \frac{\sinh(D) \cdot \prod_m (B_{2m})}{(S + \sinh(D)) \cdot \prod_m (S^2 + B_{1m} S + B_{2m})}, \tag{2.33b}$$

$$m = 0, 1, \ldots, [(n-1)/2] - 1 \ \ (n \text{ odd})$$

It is again time to consider some numerical examples before using WFilter to determine the filter coefficients.

Example 2.4 Chebyshev Third-Order Normalized Transfer Function

Problem: Determine the order, pole locations, and coefficients of the transfer function for a Chebyshev filter to satisfy the following specifications:

$a_{\text{pass}} = -1$ dB, $a_{\text{stop}} = -22$ dB, $\omega_{\text{pass}} = 1$ rad/sec, and $\omega_{\text{stop}} = 2$ rad/sec

Solution: First, we determine the fundamental constants needed from (2.20), (2.22), and (2.23):

$\varepsilon = 0.508847$	$n = 2.96$ (3rd order)	
$D = 0.475992$	$\cosh(D) = 1.115439$	$\sinh(D) = 0.494171$

Next, we find the locations of the first-order pole and the complex pole in the second quadrant from (2.24)–(2.27). A plot of the poles is shown in Figure 2.9:

(1st order)	$\sigma_R = -0.494171$	$\omega_R = 0.0$
$\phi_0 = 1\pi/6$	$\sigma_0 = -0.247085$	$\omega_0 = +0.965999$

Finally, we generate the transfer function from (2.28)–(2.33). The results from WFilter are shown in Figure 2.10.

$$H_{C,3}(s) = \frac{0.49417 \cdot 0.99420}{(S + 0.49417) \cdot (S^2 + 0.49417 \cdot S + 0.99420)}$$

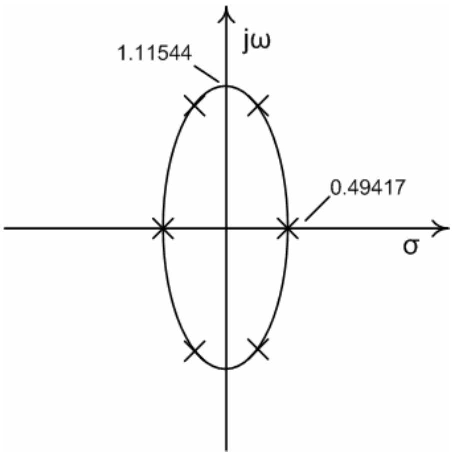
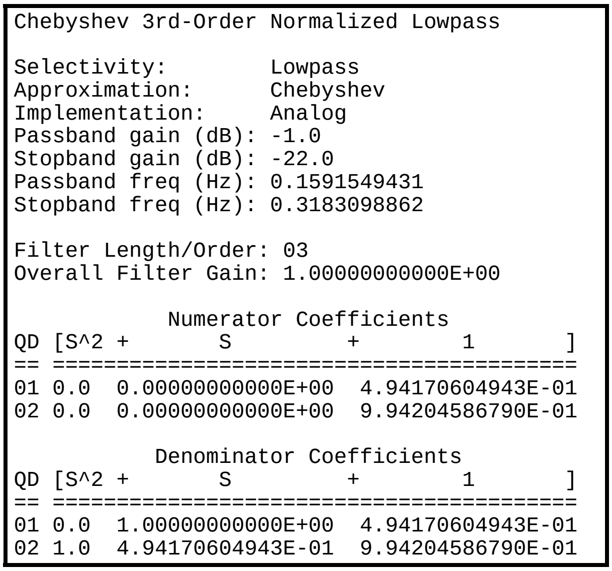


Figure 2.9 Pole locations for third-order Chebyshev normalized filter.



```
Chebyshev 3rd-Order Normalized Lowpass

Selectivity:        Lowpass
Approximation:      Chebyshev
Implementation:     Analog
Passband gain (dB): -1.0
Stopband gain (dB): -22.0
Passband freq (Hz): 0.1591549431
Stopband freq (Hz): 0.3183098862

Filter Length/Order: 03
Overall Filter Gain: 1.00000000000E+00

            Numerator Coefficients
QD [S^2 +       S         +       1       ]
== =======================================
01 0.0  0.00000000000E+00  4.94170604943E-01
02 0.0  0.00000000000E+00  9.94204586790E-01

           Denominator Coefficients
QD [S^2 +       S         +       1       ]
== =======================================
01 0.0  1.00000000000E+00  4.94170604943E-01
02 1.0  4.94170604943E-01  9.94204586790E-01
```

Figure 2.10 Chebyshev normalized third-order coefficients from WFilter.

Example 2.5 Chebyshev Fourth-Order Normalized Transfer Function

Problem: Determine the order, pole locations, and transfer function coefficients for a Chebyshev filter to satisfy the following specifications:

$$a_{\text{pass}} = -1 \text{ dB},\ a_{\text{stop}} = -33 \text{ dB},\ \omega_{\text{pass}} = 1 \text{ rad/sec, and } \omega_{\text{stop}} = 2 \text{ rad/sec}$$

Solution: First, we determine the fundamental constants needed from (2.20), (2.22), and (2.23):

$\varepsilon = 0.508847$	$n = 3.92$ (4th order)	
$D = 0.356994$	$\cosh(D) = 1.064402$	$\sinh(D) = 0.364625$

Next, we find the locations of the two complex poles in the second quadrant from (2.24)–(2.27). A plot of the poles is shown in Figure 2.11.

$\theta_0 = 1\pi/8$	$\sigma_0 = -0.139536$	$\omega_0 = +0.983379$
$\theta_1 = 3\pi/8$	$\sigma_1 = -0.336870$	$\omega_1 = +0.407329$

Finally, we generate the transfer function from (2.28)–(2.33). Note that in this even-order case, the gain constant of 0.891251 is included. The results from WFilter for this Chebyshev specification are shown in Figure 2.12.

$$H_{C,4}(S) = \frac{0.89125 \cdot 0.98650 \cdot 0.27940}{(S^2 + 0.27907 \cdot S + 0.98650) \cdot (S^2 + 0.67374 \cdot S + 0.27940)}$$

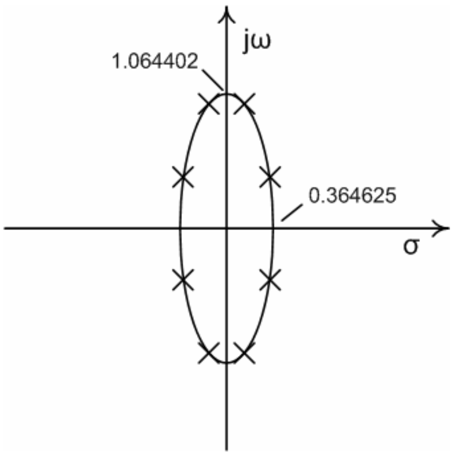


Figure 2.11 Pole locations for fourth-order Chebyshev normalized filter.

The magnitude and phase responses for the third and fourth-order Chebyshev filters are shown in Figures 2.13 and 2.14.

```
Chebyshev 4th-Order Normalized Lowpass

Selectivity:        Lowpass
Approximation:      Chebyshev
Implementation:     Analog
Passband gain (dB): -1.0
Stopband gain (dB): -33.0
Passband freq (Hz): 0.1591549431
Stopband freq (Hz): 0.3183098862

Filter Length/Order: 04
Overall Filter Gain: 8.91250938134E-01

            Numerator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 0.0  0.00000000000E+00  9.86504875318E-01
02 0.0  0.00000000000E+00  2.79398094130E-01

           Denominator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 1.0  2.79071991811E-01  9.86504875318E-01
02 1.0  6.73739387509E-01  2.79398094130E-01
```

Figure 2.12 Chebyshev normalized fourth-order coefficients from WFilter.

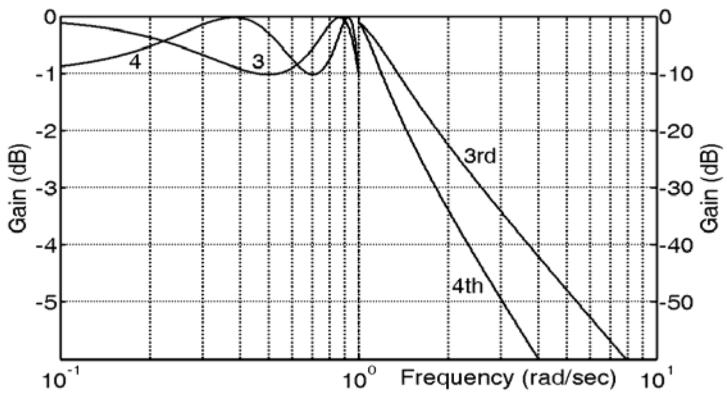


Figure 2.13 Chebyshev third-order and fourth-order magnitude responses.

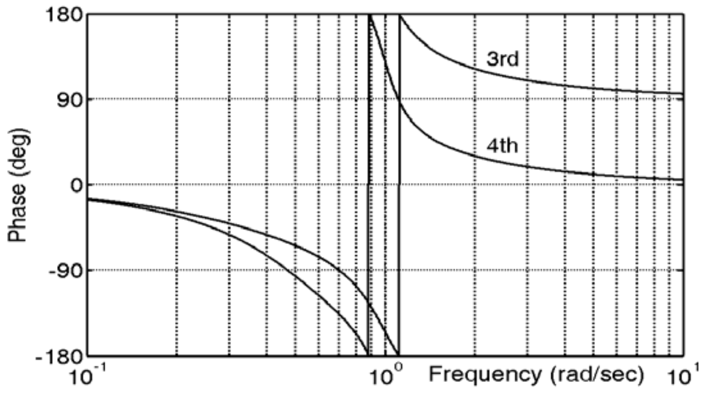


Figure 2.14 Chebyshev third-order and fourth-order phase responses.

2.4 INVERSE CHEBYSHEV NORMALIZED APPROXIMATION FUNCTIONS

The inverse Chebyshev approximation function, also called the Chebyshev type II function, is a rational approximation with both poles and zeros in its transfer function. This approximation has a smooth, maximally flat response in the passband, just as the Butterworth approximation, but has ripple in the stopband caused by the zeros of the transfer function. The inverse Chebyshev approximation provides better transition characteristics than the Butterworth filter and better phase response than the standard Chebyshev. Although the inverse Chebyshev has these features to recommend it to the filter designer, it is more involved to design.

2.4.1 Inverse Chebyshev Magnitude Response

The development of the inverse Chebyshev response is derived from the standard Chebyshev response. We will discuss the methods needed to determine the inverse Chebyshev approximation function while leaving the intricate details to the reference works. The name “inverse Chebyshev” is well-deserved in this case since we will see that many of the computations are based on inverse or reciprocal values from the standard computations. Let’s begin with the definition of the magnitude frequency response function as shown in (2.34).

The first observation concerning (2.34) is that it indeed has a numerator portion that allows for the finite zeros in the transfer function. Upon closer inspection, we find the use of ε_i in place of ε. Equation (2.35) indicates ε_i, the inverse of ε, where a_{pass} is replaced with a_{stop}. Because of the differences, we will use the subscript to distinguish ε_i from the standard ε. Although C_n still represents the Chebyshev polynomial of the first kind of degree *n* as defined in (2.21), we notice that the argument of the function is the inverse of the standard definition (ω_o/ω instead of ω/ω_o). We will see a little later in this section how these differences affect our determination of the poles and zeros of the transfer function.

$$\left| H_{I,n}\left[j(\omega/\omega_o)\right]\right| = \frac{\sqrt{\varepsilon_i^2 \cdot C_n^2(\omega_o/\omega)}}{\sqrt{1+\varepsilon_i^2 \cdot C_n^2(\omega_o/\omega)}} \tag{2.34}$$

where

$$\varepsilon_i = \frac{1}{\sqrt{10^{-0.1\cdot a_{\text{stop}}}-1}} \tag{2.35}$$

2.4.2 Inverse Chebyshev Order

Because of the nature of the derivation of the inverse Chebyshev approximation function from the standard Chebyshev approximation, it should come as no surprise that the calculation of the order for an inverse Chebyshev is the same as for the standard Chebyshev. The expression is given in (2.36) and is the same as (2.22) except for the subscript *I* designating the calculation as the inverse Chebyshev order:

$$n_I = \frac{\cosh^{-1}\left[\sqrt{(10^{-0.1\cdot a_{\text{stop}}}-1)/(10^{-0.1\cdot a_{\text{pass}}}-1)}\right]}{\cosh^{-1}(\omega_{\text{stop}}/\omega_{\text{pass}})} \tag{2.36}$$

2.4.3 Inverse Chebyshev Pole-Zero Locations

The determination of the pole locations for the normalized inverse Chebyshev approximation is based on techniques similar to those used for the standard normalized Chebyshev approximation. The pole positions for the inverse Chebyshev case are found using the same values of ϕ_m, but the value of ε_i is calculated differently. Once the pole positions are found, however, the inverse Chebyshev poles are the reciprocals of the standard poles. (There's that inverse relationship again.) For example, if there exists a standard Chebyshev pole at

$$p = \sigma + j\omega \tag{2.37}$$

then the reciprocal of p gives the inverse Chebyshev pole position as

$$p^{-1} = \frac{\sigma - j\omega}{(\sigma + j\omega)\cdot(\sigma - j\omega)} = \frac{\sigma}{\sigma^2 + \omega^2} - j\frac{\omega}{\sigma^2 + \omega^2} \tag{2.38}$$

Notice that if a pole's distance from the origin is greater than one, the reciprocal's distance will be less than one, and vice versa. In addition, the position of the pole is reflected across the real axis, so although the original pole position may be in the second quadrant, the reciprocal is located in the third quadrant. Consequently, if we are able to determine pole positions for the standard Chebyshev approximation function as discussed in the previous section, we should have little problem finding the inverse Chebyshev pole locations.

Let's derive the mathematical equations necessary to determine the pole locations for the inverse Chebyshev approximation function along the same lines as we did for the standard Chebyshev case. First, D_i will be defined in terms of ε_i in (2.39).

$$D_i = \frac{\sinh^{-1}(\varepsilon_i^{-1})}{n} \tag{2.39}$$

Next, we can define the pole locations in the second quadrant in the manner of the previous section as shown in (2.40)–(2.42), remembering that these primed values must still be inverted.

$$\sigma'_m = -\sinh(D_i)\cdot\sin(\phi_m) \tag{2.40}$$

$$\omega'_m = \cosh(D_i)\cdot\cos(\phi_m) \tag{2.41}$$

$$\phi_m = \frac{\pi\cdot(2\cdot m + 1)}{2\cdot n}, m = 0, 1, \ldots, (n/2) - 1 \ (n \text{ even}) \tag{2.42a}$$

$$\phi_m = \frac{\pi\cdot(2\cdot m + 1)}{2\cdot n}, m = 0, 1, \ldots, [(n-1)/2] - 1 \ (n \text{ odd}) \tag{2.42b}$$

We can determine the final pole locations by inverting these poles as indicated in (2.43) and (2.44):

$$\sigma_m = \frac{\sigma'_m}{\sigma'^2_m + \omega'^2_m} \tag{2.43}$$

$$\omega_m = \frac{-\omega'_m}{\sigma'^2_m + \omega'^2_m} \tag{2.44}$$

If the approximation function is odd-order, then there will be a first-order pole on the negative real axis at σ_R as defined in (2.45):

$$\sigma_R = -[\sinh(D_i)]^{-1} \tag{2.45}$$

Next, we need to determine the placement of the finite zeros of the inverse Chebyshev approximation function, which are all purely imaginary complex conjugate pairs located on the $j\omega$ axis. Because they only occur in pairs, the numerator of an inverse Chebyshev transfer function will always be even. If the order of the denominator is odd, then one zero of the transfer function will be located at infinity. The location of the zeros on the $j\omega$ axis is determined by (2.46) and (2.47), where ϕ_m is as defined in (2.42). A z is used in the subscript to differentiate the zero locations from the pole locations. (By the way, did you notice that the secant function in (2.47) is the *reciprocal* of the cosine function used in the standard Chebyshev function?)

$$\sigma_{zm} = 0.0 \tag{2.46}$$

$$\omega_{zm} = \sec(\phi_m) \tag{2.47}$$

2.4.4 Inverse Chebyshev Transfer Functions

Now that we have located the necessary poles and zeros that are pertinent to the definition of the inverse Chebyshev approximation, we can define the various factors that describe the transfer function. First, for odd-order approximations, (2.48) describes the first-order factor:

$$H_o(S) = \frac{[\sinh(D_i)]^{-1}}{S + [\sinh(D_i)]^{-1}} \tag{2.48}$$

Next, the quadratic components of the transfer function are described in (2.49)–(2.53). These are similar to the quadratic definition for the Chebyshev case, but we have added a numerator quadratic for the zeros as well.

$$H_m(S) = \frac{B_{2m} \cdot (S^2 + A_{1m} \cdot S + A_{2m})}{A_{2m} \cdot (S^2 + B_{1m} \cdot S + B_{2m})} \tag{2.49}$$

where

$$B_{1m} = -2 \cdot \sigma_m \tag{2.50}$$

$$B_{2m} = \sigma_m^2 + \omega_m^2 \tag{2.51}$$

$$A_{1m} = -2 \cdot \sigma_{zm} = 0.0 \tag{2.52}$$

$$A_{2m} = \sigma_{zm}^2 + \omega_{zm}^2 = \omega_{zm}^2 \tag{2.53}$$

Although the value of A_{1m} is zero, it is included to be consistent with the format used throughout the remainder of the text.

We are now ready to define the generalized transfer function form for the inverse Chebyshev approximation function shown in (2.54). Since the inverse Chebyshev has a maximally flat response in the passband as the Butterworth, there is no need for a gain adjustment constant as in the standard Chebyshev case.

$$H_{I,n}(S) = \frac{\prod_m (B_{2m}) \cdot \prod_m (S^2 + A_{1m} \cdot S + A_{2m})}{\prod_m (A_{2m}) \cdot \prod_m (S^2 + B_{1m} \cdot S + B_{2m})}, \tag{2.54a}$$
$$m = 0, 1, \dots, (n/2) - 1 \ \ (n \text{ even})$$

$$H_{I,n}(S) = \frac{[\sinh(D_i)]^{-1} \cdot \prod_m (B_{2m}) \cdot \prod_m (S^2 + A_{1m} \cdot S + A_{2m})}{(S + [\sinh(D_i)]^{-1}) \cdot \prod_m (A_{2m}) \cdot \prod_m (S^2 + B_{1m} \cdot S + B_{2m})}, \tag{2.54b}$$
$$m = 0, 1, \dots, [(n-1)/2] - 1 \ \ (n \text{ odd})$$

The following numerical examples should help to illustrate the process.

<u>Example 2.6 Inverse Chebyshev Third-Order Normalized Transfer Function</u>

Problem: Determine the order, pole and zero locations, and transfer function coefficients for an inverse Chebyshev filter to satisfy the following specifications:

$a_{\text{pass}} = -1$ dB, $a_{\text{stop}} = -22$ dB, $\omega_{\text{pass}} = 1$ rad/sec, and $\omega_{\text{stop}} = 2$ rad/sec

Solution: First, we determine the fundamental constants needed from (2.35), (2.36), and (2.39):

$\varepsilon_i = 0.079685$	$n = 2.96$ (3rd order)	
$D_i = 1.074803$	$\cosh(D_i) = 1.635391$	$\sinh(D_i) = 1.294026$

Next, we find the locations of the first-order pole, the complex pole in the second quadrant, and the second-order zeros on the $j\omega$ axis from (2.40)–(2.47). A pole-zero plot is shown in Figure 2.15.

(1st order)	$\sigma_R = -0.772782$	$\omega_R = 0.0$
$\phi_0 = 1\pi/6$	$\sigma'_0 = -0.647013$	$\omega'_0 = +1.416290$
	$\sigma_0 = -0.266864$	$\omega_0 = -0.584157$
(zeros)	$\sigma_{z0} = +0.0$	$\omega_{z0} = +1.154701$

Finally, we generate the transfer function from (2.48)–(2.54):

$$H^*_{I,3}(S) = \frac{0.77278 \cdot 0.41246 \cdot (S^2 + 1.3333)}{1.3333 \cdot (S + 0.77278) \cdot (S^2 + 0.53373 \cdot S + 0.41246)}$$

$$H_{I,3}(S) = \frac{0.30934 \cdot 1.5456 \cdot (S^2 + 5.3333)}{(S + 1.5456) \cdot (S^2 + 1.0675 \cdot S + 1.6498)}$$

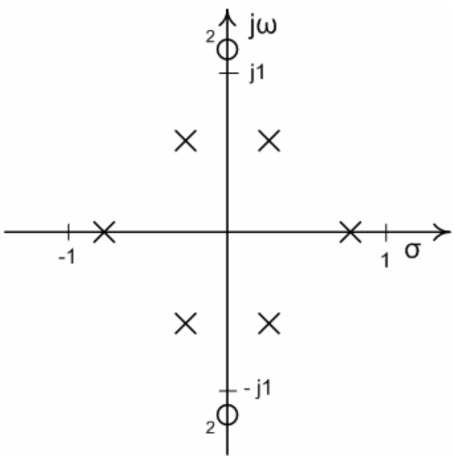


Figure 2.15 Pole and zero locations for third-order inverse Chebyshev filter.

There is a problem with the first transfer function above (shown with an asterisk *). It implements an inverse Chebyshev approximation function that is normalized to $\omega_{\text{stop}} = 1$ rad/sec instead of $\omega_{\text{pass}} = 1$ rad/sec. (This means that ω_{pass}

would be at 0.5 rad/sec.) Therefore the entire frequency response is a factor of 2 too low. The attenuation at ω = 0.5 rad/sec is ~ 1 dB and the attenuation at ω = 1 rad/sec is ~22 dB. The process we use to correct the problem is actually an unnormalization procedure that is covered in Chapter 3. This unnormalization will usually occur as part of the total filter design process, but we can make the adjustment manually in this particular case. The correct transfer function can be determined by substituting *S*/2 for *S* and then simplifying as indicated in the second transfer function above. This process is mentioned here so we understand the WFilter coefficients, which are shown in Figure 2.16.

```
Inv. Chebyshev 3rd-Order Normal. Lowpass

Selectivity:        Lowpass
Approximation:      Inv. Chebyshev
Implementation:     Analog
Passband gain (dB): -1.0
Stopband gain (dB): -22.0
Passband freq (Hz): 0.1591549431
Stopband freq (Hz): 0.3183098862

Filter Length/Order: 03
Overall Filter Gain: 3.09341803036E-01

            Numerator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 0.0  0.00000000000E+00  1.54556432589E+00
02 1.0  0.00000000000E+00  5.33333333334E+00

           Denominator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 0.0  1.00000000000E+00  1.54556432589E+00
02 1.0  1.06745667061E+00  1.64982294953E+00
```

Figure 2.16 Inverse Chebyshev normalized third-order coefficients from WFilter.

Example 2.7 Inverse Chebyshev Fourth-Order Normalized Transfer Function

Problem: Determine the order, pole and zero locations, and transfer function coefficients for an inverse Chebyshev filter to satisfy the following specifications:

$$a_{\text{pass}} = -1 \text{ dB},\ a_{\text{stop}} = -33 \text{ dB},\ \omega_{\text{pass}} = 1 \text{ rad/sec, and } \omega_{\text{stop}} = 2 \text{ rad/sec}$$

Solution: First, we determine the fundamental constants needed from (2.35), (2.36), and (2.39):

$\varepsilon_i = 0.022393$ $n = 3.92$ (4th order)

$D_i = 1.123072$ $\cosh(D_i) = 1.699781$ $\sinh(D_i) = 1.374502$

Next, we find the locations of the two complex poles in the second quadrant and the second-order zeros from (2.40)–(2.47). A pole-zero plot is shown in Figure 2.17.

$\phi_0 = 1\pi/8$	$\sigma'_0 = -0.525999$	$\omega'_0 = +1.570393$
	$\sigma_0 = -0.191774$	$\omega_0 = -0.572549$
$\phi_0 = 3\pi/8$	$\sigma'_0 = -1.269874$	$\omega'_0 = +0.650478$
	$\sigma_0 = -0.623801$	$\omega_0 = -0.319535$
(Zeros)	$\sigma_{z0} = +0.0$	$\omega_{z0} = +1.082392$
(Zeros)	$\sigma_{z1} = +0.0$	$\omega_{z1} = +2.613126$

Finally, we generate the transfer function from (2.48)–(2.54). (Refer to Example 2.6 for an explanation of the two transfer functions.) The WFilter coefficients are shown in Figure 2.18.

$$H^*_{I,4}(S) = \frac{0.022387 \cdot (S^2 + 1.1716) \cdot (S^2 + 6.8284)}{(S^2 + 0.38355 \cdot S + 0.36459) \cdot (S^2 + 1.2476 \cdot S + 0.49123)}$$

$$H_{I,4}(S) = \frac{0.022387 \cdot (S^2 + 4.6863) \cdot (S^2 + 27.314)}{(S^2 + 0.76710 \cdot S + 1.4584) \cdot (S^2 + 2.4952 \cdot S + 1.9649)}$$

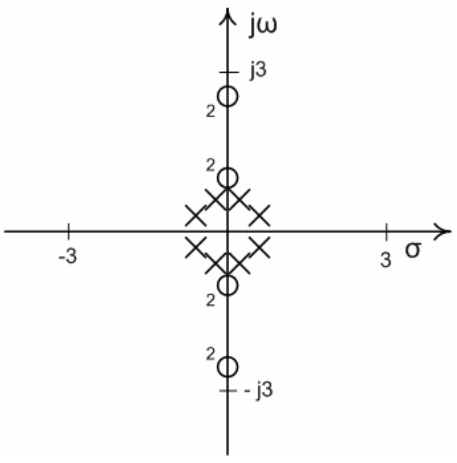


Figure 2.17 Pole and zero locations for fourth-order inverse Chebyshev filter.

```
Inv. Chebyshev 4th-Order Normal. Lowpass

Selectivity:        Lowpass
Approximation:      Inv. Chebyshev
Implementation:     Analog
Passband gain (dB): -1.0
Stopband gain (dB): -33.0
Passband freq (Hz): 0.1591549431
Stopband freq (Hz): 0.3183098862

Filter Length/Order: 04
Overall Filter Gain: 2.23872113857E-02

            Numerator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 1.0  0.00000000000E+00  4.68629150102E+00
02 1.0  0.00000000000E+00  2.73137084990E+01

           Denominator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 1.0  7.67095479088E-01  1.45835864853E+00
02 1.0  2.49520593780E+00  1.96492341597E+00
```

Figure 2.18 Inverse Chebyshev normalized fourth-order coefficients.

The magnitude and phase responses for the two inverse Chebyshev examples are presented in Figures 2.19 and 2.20. The same procedure was used as in the Butterworth and Chebyshev cases.

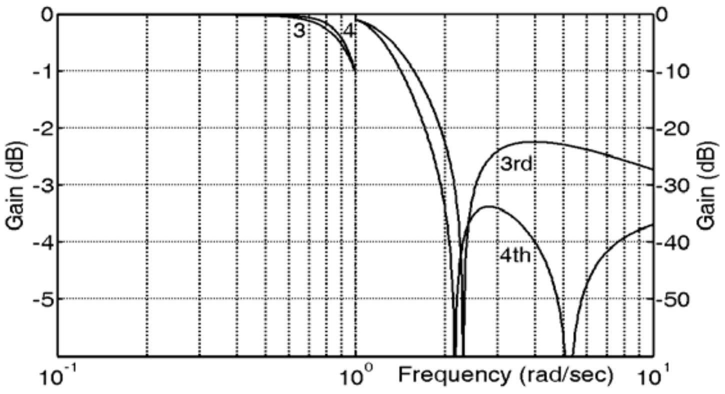


Figure 2.19 Inverse Chebyshev third-order and fourth-order magnitude responses.

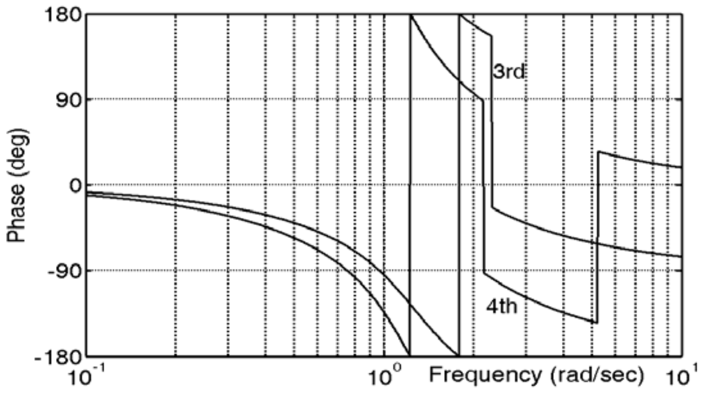


Figure 2.20 Inverse Chebyshev third-order and fourth-order phase responses.

2.5 ELLIPTIC NORMALIZED APPROXIMATION FUNCTIONS

The elliptic or Cauer approximation function provides the best selectivity characteristic of any of the approximation methods discussed thus far. No other approximation method will be able to provide a lower-order filter for the specifications provided. The elliptic filter combines ripple in the passband and stopband in order to accomplish this feat. However, the elliptic approximation is also the most difficult to design. It involves the most sophisticated mathematical functions of any of the methods discussed in this text. Luckily, many good minds have laid the foundation for this work and their results will be presented here so that we can put the design procedure into a workable algorithm.

2.5.1 Elliptic Magnitude Response

The elliptic approximation's magnitude frequency response function is shown in (2.55), where R_n is the Chebyshev rational function of order n. R_n is composed of both numerator and denominator portions, which allow an equiripple response in both the passband and stopband.

The Chebyshev rational function R_n and much of elliptic approximation theory is based on the elliptic integral and the Jacobian elliptic functions. These functions can be evaluated via advanced mathematical packages available for most computers and are discussed in Appendix D. The incomplete elliptic integral of the first kind is shown in (2.57), where k is referred to as the modulus and ϕ is the amplitude of the integral. The modulus k must be less than or equal to 1 for the elliptic integral to be real. The elliptic sine, cosine, tangent, and difference functions based on the elliptic integral are given in (2.58)–(2.61), respectively.

These functions are used in the calculation of the pole-zero locations in the next section.

$$\left|H_{E,n}[j(\omega/\omega_o)]\right| = \frac{1}{\sqrt{1+\varepsilon^2 \cdot R_n^2(\omega_o/\omega)}} \tag{2.55}$$

where ε is as defined previously.

$$\varepsilon = \sqrt{10^{-0.1 \cdot a_{\text{pass}}} - 1} \tag{2.56}$$

$$u(\phi,k) = \int_0^{\phi} (1-k^2 \sin^2 x)^{-1/2}\, dx \tag{2.57}$$

$$sn(u,k) = \sin(\phi) \tag{2.58}$$

$$cn(u,k) = \cos(\phi) \tag{2.59}$$

$$sc(u,k) = \tan(\phi) \tag{2.60}$$

$$dn(u,k) = \frac{d\phi}{du} \tag{2.61}$$

The complete elliptic integral of the first kind will be used more often than the incomplete integral and it is defined in (2.62). It should be noted at this point that there are various ways to define the elliptic integrals and elliptic functions. Some authors use the modulus *k* as we have in this text, while others use other parameters related to *k*.

$$CEI(k) = u(\pi/2,k) = \int_0^{\pi/2} (1-k^2 \sin^2 x)^{-1/2}\, dx \tag{2.62}$$

2.5.2 Elliptic Order

The order of the elliptic approximation function required to meet the specifications for a filter is given in (2.63):

$$n_E = \frac{CEI(rt)\cdot CEI(\sqrt{1-kn^2})}{CEI(\sqrt{1-rt^2})\cdot CEI(kn)} \tag{2.63}$$

where *CEI* refers to the complete elliptic integral, and the ratio *rt* and the kernel *kn* are defined as

$$rt = \omega_{\text{pass}} / \omega_{\text{stop}} \tag{2.64}$$

$$kn = \sqrt{(10^{-0.1a_{\text{pass}}} - 1)/(10^{-0.1a_{\text{stop}}} - 1)} \tag{2.65}$$

Example 2.8 Elliptic Order Calculation

Problem: Determine the order of an elliptic filter required to satisfy the following specifications:

$a_{\text{pass}} = -1$ dB, $a_{\text{stop}} = -34$ dB, $\omega_{\text{pass}} = 1$ rad/sec, and $\omega_{\text{stop}} = 2$ rad/sec

Solution: In order to determine the order of the elliptic approximation, we first determine that $rt = 0.5$ and $kn = 0.0101548$. Then, using any appropriate math package, we can determine that

$$n_E = \frac{1.686 \cdot 5.976}{2.157 \cdot 1.571} = 2.97$$

which indicates that a third-order filter will be required. Notice that the standard and inverse Chebyshev approximations require a fourth-order function to provide $a_{\text{stop}} = -33$ dB and a Butterworth approximation would require a seventh-order function to meet this specification.

2.5.3 Elliptic Pole-Zero Locations

The pole and zero locations for the elliptic approximation function are also dependent on the elliptic integral and the elliptic functions defined in the previous section. We'll start by defining a variable v_o, which is used in the calculation of the pole and zero locations.

$$v_o = \frac{CEI(rt) \cdot sc^{-1}(\varepsilon^{-1}, kn)}{n \cdot CEI(kn)} \tag{2.66}$$

Next, the pole's real and imaginary components are determined as

$$\sigma_m = -\frac{cn[f(m), rt] \cdot dn[f(m), rt] \cdot sn\left(v_o, \sqrt{1-rt^2}\right) \cdot cn\left(v_o, \sqrt{1-rt^2}\right)}{1 - dn^2[f(m), rt] \cdot sn^2\left(v_o, \sqrt{1-rt^2}\right)} \tag{2.67}$$

$$\omega_m = \frac{sn[f(m), rt] \cdot dn\left(v_o, \sqrt{1-rt^2}\right)}{1 - dn^2[f(m), rt] \cdot sn^2\left(v_o, \sqrt{1-rt^2}\right)} \tag{2.68}$$

where

$$f(m) = \frac{CEI(rt) \cdot (2 \cdot m + 1)}{n}, m = 0, 1, \ldots, (n/2) - 1 \ (n \text{ even}) \tag{2.69a}$$

$$f(m) = \frac{CEI(rt) \cdot (2 \cdot m + 2)}{n}, m = 0, 1, \ldots, [(n-1)/2] - 1 \ (n \text{ odd}) \tag{2.69b}$$

Note the negative sign for σ_m, which effectively moves the pole location from the first quadrant to the second quadrant.

In the case of odd-order approximations, the first-order denominator pole will be located on the negative real axis at

$$\sigma_R = -\frac{sn\left(v_o, \sqrt{1-rt^2}\right) \cdot cn\left(v_o, \sqrt{1-rt^2}\right)}{1 - sn^2\left(v_o, \sqrt{1-rt^2}\right)} \tag{2.70}$$

And finally, the location of the zeros that will be purely imaginary on the $j\omega$ axis are given by

$$\sigma_{zm} = 0.0 \tag{2.71}$$

$$\omega_{zm} = \frac{1}{rt \cdot sn\left[f(m), rt\right]} \tag{2.72}$$

Although the elliptic approximation requires a number of mathematical functions which aren't in everyday usage, we have most of the hard work done in determining the transfer function we need. Our primary objective in this section is to develop an orderly manner to calculate the pole and zero locations.

2.5.4 Elliptic Transfer Functions

Now we are able to define the first-order and quadratic factors that will make up the elliptic approximation function. The first-order factor for the elliptic approximation is indicated in (2.73), where σ_R is as indicated in (2.70). Again, there is no matching finite zero for the first-order pole factor; it is located at infinity.

$$H_o(S) = \frac{\sigma_R}{S + \sigma_R} \tag{2.73}$$

The form of the quadratic components of the transfer function will also be identical to the inverse Chebyshev case, as indicated below:

$$H_m(S) = \frac{B_{2m} \cdot (S^2 + A_{1m} \cdot S + A_{2m})}{A_{2m} \cdot (S^2 + B_{1m} \cdot S + B_{2m})} \tag{2.74}$$

where

$$B_{1m} = -2 \cdot \sigma_m \tag{2.75}$$

$$B_{2m} = \sigma_m^2 + \omega_m^2 \tag{2.76}$$

$$A_{1m} = -2 \cdot \sigma_{zm} = 0.0 \tag{2.77}$$

$$A_{2m} = \sigma_{zm}^2 + \omega_{zm}^2 = \omega_{zm}^2 \tag{2.78}$$

We are now ready to define a generalized transfer function for the elliptic approximation function that is almost identical to the inverse Chebyshev case. The difference lies in the ripple in the passband as in the standard Chebyshev case.

Consequently, the even-order ripple adjustment factor is included in (2.79). The ratio of product factors is combined with this value to determine the total gain adjustment:

$$H_{E,n}(S) = \frac{(10^{0.05 \cdot a_{\text{pass}}}) \cdot \prod_m (B_{2m}) \cdot \prod_m (S^2 + A_{1m} \cdot S + A_{2m})}{\prod_m (A_{2m}) \cdot \prod_m (S^2 + B_{1m} \cdot S + B_{2m})}, \qquad (2.79a)$$

$$m = 0, 1, \ldots, (n/2) - 1 \ (n \text{ even})$$

$$H_{E,n}(S) = \frac{\sigma_R \cdot \prod_m (B_{2m}) \cdot \prod_m (S^2 + A_{1m} \cdot S + A_{2m})}{(S + \sigma_R) \cdot \prod_m (A_{2m}) \cdot \prod_m (S^2 + B_{1m} \cdot S + B_{2m})}, \qquad (2.79b)$$

$$m = 0, 1, \ldots, [(n-1)/2] - 1 \ (n \text{ odd})$$

<u>Example 2.9 Elliptic Third-Order Normalized Transfer Function</u>

Problem: Determine the order, pole and zero locations, and transfer function coefficients for an elliptic filter to satisfy the following specifications:

a_{pass} = –1 dB, a_{stop} = –34 dB, ω_{pass} = 1 rad/sec, and ω_{stop} = 2 rad/sec

Solution: First, we determine the fundamental constants needed from (2.56) and (2.63)–(2.66):

ε = 0.508847	n = 2.97 (3rd order)
rt = 0.50	kn = 0.0101549
CEI(rt) = 1.685750	CEI(kn) = 1.570837
CEI[sqrt(1 – rt^2)] = 2.156516	CEI[sqrt(1 – kn^2)] = 5.976226
v_o = 0.510786	

Next, we find the locations of the first-order pole, the complex pole in the second quadrant, and the second-order zeros on the $j\omega$ axis from (2.67)–(2.72). A pole-zero plot is shown in Figure 2.21.

(1st order)	σ_R = –0.539953	ω_R = 0.0
$f(0)$ = 1.123834	σ_0 = –0.217032	ω_0 = +0.981574
(Zeros)	σ_{z0} = +0.0	ω_{z0} = +2.270068

Finally, we generate the transfer function from (2.73)–(2.79). The WFilter coefficients are given in Figure 2.22.

$$H_{E,3}(s) = \frac{0.53995 \cdot 1.0106 \cdot (S^2 + 5.1532)}{5.1532 \cdot (S + 0.53995) \cdot (S^2 + 0.43406 \cdot S + 1.0106)}$$

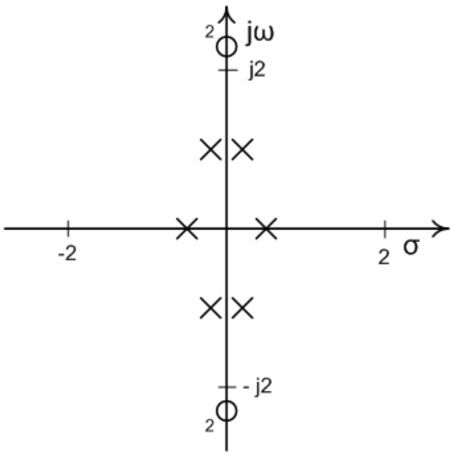


Figure 2.21 Pole and zero locations for third-order elliptic normalized filter.

```
Elliptic 3rd-Order Normalized Lowpass

Selectivity:         Lowpass
Approximation:       Elliptic
Implementation:      Analog
Passband gain (dB): -1.0
Stopband gain (dB): -34.0
Passband freq (Hz): 0.1591549431
Stopband freq (Hz): 0.3183098862

Filter Length/Order: 03
Overall Filter Gain: 1.96108842659E-01

            Numerator Coefficients
QD [S^2 +       S          +        1       ]
== =========================================
01 0.0  0.00000000000E+00  5.39953773543E-01
02 1.0  0.00000000000E+00  5.15320911642E+00

           Denominator Coefficients
QD [S^2 +       S          +        1       ]
== =========================================
01 0.0  1.00000000000E+00  5.39953773543E-01
02 1.0  4.34064063925E-01  1.01058987580E+00
```

Figure 2.22 Elliptic normalized third-order coefficients from WFilter.

Example 2.10 Elliptic Fourth-Order Normalized Transfer Function

Problem: Determine the order, pole and zero locations, and transfer function coefficients for an elliptic filter to satisfy the following specifications:

$a_{\text{pass}} = -1$ dB, $a_{\text{stop}} = -51$ dB, $\omega_{\text{pass}} = 1$ rad/sec, and $\omega_{\text{stop}} = 2$ rad/sec

Solution: First, we determine the fundamental constants needed:

$\varepsilon = 0.508847$	$n = 3.95$ (4th order)
$rt = 0.50$	$kn = 0.00143413$
$\text{CEI}(rt) = 1.685750$	$\text{CEI}(kn) = 1.570797$
$\text{CEI}[\text{sqrt}(1 - rt^2)] = 2.156516$	$\text{CEI}[\text{sqrt}(1 - kn^2)] = 7.933494$
$vo = 0.383119$	

Next, we find the locations of the two complex poles in the second quadrant and the second-order zeros on the jω axis from (2.67)–(2.72). A pole-zero plot is shown in Figure 2.23.

$f(0) = 0.421438$	$\sigma_0 = -0.351273$	$\omega_0 = +0.442498$
$f(1) = 1.264313$	$\sigma_1 = -0.121478$	$\omega_1 = +0.989176$
(Zeros)	$\sigma_{z0} = +0.0$	$\omega_{z0} = +4.922113$
(Zeros)	$\sigma_{z0} = +0.0$	$\omega_{z0} = +2.143189$

Finally, we generate the transfer function from (2.73)–(2.79). Note that in this even-order case, the gain constant of 0.891251 is included. The WFilter coefficients are given in Figure 2.24.

$$H_{E,4}(S) = \frac{0.0025391 \cdot (S^2 + 24.227) \cdot (S^2 + 4.5933)}{(S^2 + 0.70255 \cdot S + 0.31920) \cdot (S^2 + 0.24296 \cdot S + 0.99323)}$$

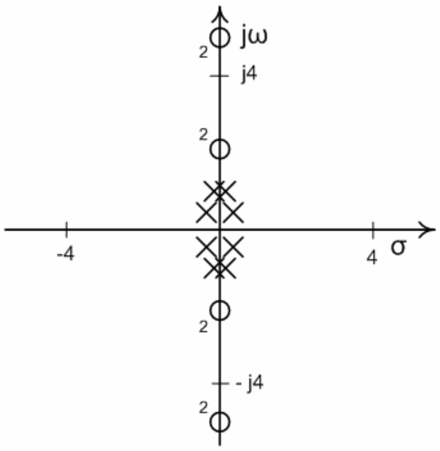


Figure 2.23 Pole and zero locations for fourth-order elliptic normalized filter.

```
Elliptic 4th-Order Normalized Lowpass

Selectivity:        Lowpass
Approximation:      Elliptic
Implementation:     Analog
Passband gain (dB): -1.0
Stopband gain (dB): -51.0
Passband freq (Hz): 0.1591549431
Stopband freq (Hz): 0.3183098862

Filter Length/Order: 04
Overall Filter Gain: 2.53911536581E-03

            Numerator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 1.0  0.00000000000E+00  2.42272011683E+01
02 1.0  0.00000000000E+00  4.59326052578E+00

           Denominator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 1.0  7.02545661306E-01  3.19196825769E-01
02 1.0  2.42956737746E-01  9.93226261783E-01
```

Figure 2.24 Elliptic normalized fourth-order coefficients from WFilter.

The magnitude and phase responses for the elliptic filters are presented in Figures 2.25 and 2.26.

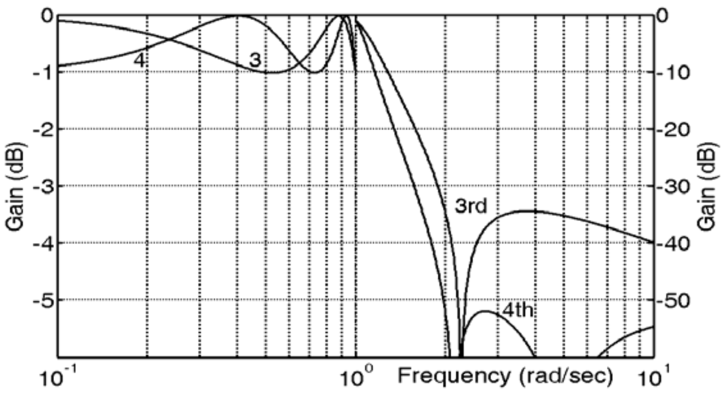


Figure 2.25 Elliptic third-order and fourth-order magnitude responses.

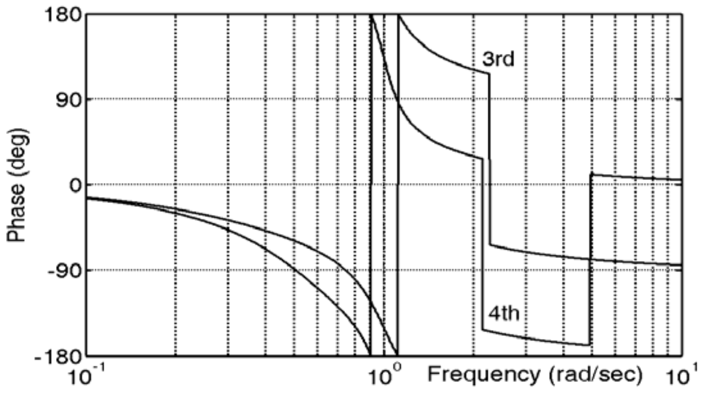


Figure 2.26 Elliptic third-order and fourth-order phase responses.

2.6 COMPARISON OF APPROXIMATION METHODS

Now that we have discussed the four approximation methods and displayed third-order and fourth-order magnitude and phase plots, we are in a position to compare the results. First, we look at the magnitude plots of Figures 2.7, 2.13, 2.19, and 2.25. Table 2.1 shows the gains achieved at the stopband edge frequency of 2 rad/sec for each normalized filter type and order. (Each filter was designed with a passband gain of –1 dB.) Obviously, if attenuation characteristics in the stopband are the primary concern, an elliptic filter would have to be the choice.

It provides 12 dB more attenuation than the Chebyshev types and 22 dB more attenuation than the Butterworth filter for the third-order case. In the fourth-order case, the differences increase to over 18 and 33 dB compared to the Chebyshev and Butterworth filters. The Chebyshev filter types themselves afford better stopband characteristics when compared to the Butterworth filter. They provide 10 and 15 dB more attenuation for the third-order and fourth-order cases. Although the table only lists the gains for third-order and fourth-order filters, the same trend continues for higher-order filters.

Although the Chebyshev and inverse Chebyshev filters provide the same gains at the passband and stopband edge frequencies, their responses are not identical. If we were to take a close look at the frequency response in the passband, we would find that the inverse Chebyshev provides a better approximation to the ideal response except at frequencies very near to 1 (the normalized passband edge frequency). In that case, the standard Chebyshev produces a tighter fit. In the transition band, the standard Chebyshev response provides a more rapid transition. And in the stopband, the standard Chebyshev’s response continues to increase the attenuation as the frequency increases, while the inverse Chebyshev’s response alternates between small gains and a_{stop}. In some cases, the filter designer might trade the faster transition for the nondecreasing attenuation.

Table 2.1
Comparison of Filter Gains at 2 rad/sec

Filter Type	*3rd Order*	*4th Order*
Butterworth	–12.5 dB	–18.3 dB
Chebyshev	–22.5 dB	–33.8 dB
Inverse Chebyshev	–22.5 dB	–33.8 dB
Elliptic	–34.5 dB	–51.9 dB

Although the magnitude characteristics of a filter are very important, the phase characteristics of a filter are also crucial in many projects. Whether in audio networks or data transmission systems, designers are looking for filters with linear phase response. Nonlinear phase response in an audio network will cause noticeable phase distortion for the listener that cannot be tolerated, especially in high-quality systems. In data transmission systems nonlinear phase response produces group delays that are functions of frequency. This produces distortion in the pulses sent over the system and can distort edges and levels to the point of causing errors in the received signal. We can compare the phase responses of Figures 2.8, 2.14, 2.20, and 2.26 to see the level of phase distortion for each approximation type. Remember that the transitions from –180 to +180 degrees are not discontinuities, but rather a function of the display method. (The phase response is written to a data file in its true form.) Table 2.2 shows the phase angles for the third-order and fourth-order filters at the passband and stopband edge frequencies.

Table 2.2
Comparison of Filter Phase at 1 and 2 rad/sec

Filter Type	*3rd @ 1 r/s*	*3rd @ 2 r/s*	*4th @ 1 r/s*	*4th @ 2 r/s*
Butterworth	–104°	–192°	–146°	–266°
Chebyshev	–154°	–238°	–230°	–330°
Inverse Chebyshev	–94°	–192°	–133°	–264°
Elliptic	–150°	–238°	–226°	–330°

As Table 2.2 and the phase plots indicate, the filters with the maximally flat response in the passband (Butterworth and inverse Chebyshev) provide the most linear response, although the inverse Chebyshev does have phase discontinuities in the stopband caused by the complex zeros. These are usually not critical because the filter's magnitude response is very small at these frequencies and the distortion should be minimal. The phase responses of the standard Chebyshev and elliptic are also matched very closely and can be judged equivalent except for the discontinuities in the stopband caused by the zeros for the elliptic case.

A filter designer's task is not always clear cut. It seems that every project requires as much stopband attenuation as possible while providing a phase response as linear as possible. The task becomes one of weighing the importance of each characteristic. If phase response is more critical than magnitude response, then the Butterworth filter is a better choice. If the opposite is true, the elliptic filter is a better choice. If magnitude and phase responses are nearly equal in importance, then one of the Chebyshev filters may be the best choice. Other alternatives are also possible. Elliptic filters can be used for their selectivity, with phase compensation filters added to make the phase more linear. (These filter types are not covered in this text, but references in the analog filter design section of Appendix A provide further information.) A designer must be careful when pursuing these alternatives, since in some cases the result may be no better than the equivalent Butterworth or Chebyshev filter.

2.7 CONCLUSION

In this chapter, we studied the core of analog filter design, the normalized approximation functions. By developing these functions, we have laid the foundation for the remainder of the chapters on analog filter design as well as a good bit of digital IIR filter design. By approaching each approximation function in the same manner, and developing methods for determining exact pole and zero placement, we have simplified the job of generating the C code necessary to implement these algorithms in a clean, efficient manner. (Those who are interested in seeing more on the development of the C code can turn to Appendix D.) In the next chapter, we will finish up the analog filter design calculations by determining a technique to unnormalize the transfer functions we have just developed.

Chapter 3

Analog Lowpass, Highpass, Bandpass, and Bandstop Filters

In the last chapter, we were able to determine the normalized approximation functions for the most common types of analog filters. Our task in this chapter is to unnormalize those approximation functions in a manner to produce lowpass, highpass, bandpass, and bandstop filters at the desired frequencies. This unnormalization will be carried out in such a way that the design of the normalized approximation functions will be central to the development. Figure 3.1 shows the three-step procedure used in the unnormalization. The simplicity of this procedure is the fact that the second step is the same for all filter design methods.

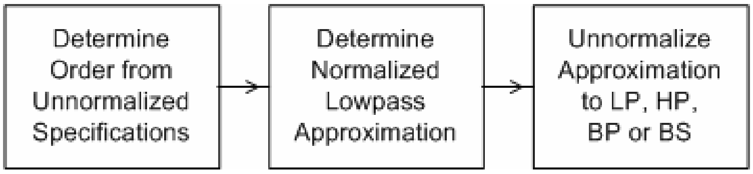

Determine Order from Unnormalized Specifications
Determine Normalized Lowpass Approximation
Unnormalize Approximation to LP, HP, BP or BS

Figure 3.1 Procedure for unnormalization.

3.1 UNNORMALIZED LOWPASS APPROXIMATION FUNCTIONS

Even though the normalized approximation functions determined in the previous chapter are lowpass functions, they still need to be unnormalized to the proper operational frequency. The first step in the unnormalization procedure, as indicated in Figure 3.1, is to determine the order of the approximation function from the unnormalized specifications. The order of approximation function depends only on the passband and stopband gains and frequencies. The gains for both the normalized and unnormalized approximation functions will be the same, the only specifications that change are the passband and stopband edge frequencies. However, as indicated in Chapter 2, it is not the individual frequencies that determine the order of the approximation function, but rather the ratio of the frequencies. Therefore, we can define a frequency ratio variable in (3.1) that will be used for the lowpass filter type as indicated by the additional subscript *L*.

$$\Omega_{rL} = \frac{\omega_{\text{stop}}}{\omega_{\text{pass}}} = \frac{f_{\text{stop}}}{f_{\text{pass}}} \tag{3.1}$$

Each of the equations from Chapter 2 that were used to determine the order of a particular filter type can now be redefined in terms of Ω_r, as indicated in (3.2)–(3.6).

$$n_B = \frac{\log[(10^{-0.1\cdot a_{\text{stop}}} - 1)/(10^{-0.1\cdot a_{\text{pass}}} - 1)]}{2\cdot\log(\Omega_r)} \tag{3.2}$$

$$n_C = n_I = \frac{\cosh^{-1}\left[\sqrt{(10^{-0.1\cdot a_{\text{stop}}} - 1)/(10^{-0.1\cdot a_{\text{pass}}} - 1)}\right]}{\cosh^{-1}(\Omega_r)} \tag{3.3}$$

$$n_E = \frac{CEI(rt)\cdot CEI\left(\sqrt{1-kn^2}\right)}{CEI\left(\sqrt{1-rt^2}\right)\cdot CEI(kn)} \tag{3.4}$$

where

$$rt = 1/\Omega_r \tag{3.5}$$

$$kn = \sqrt{(10^{-0.1a_{\text{pass}}} - 1)/(10^{-0.1a_{\text{stop}}} - 1)} \tag{3.6}$$

It may appear that we are doing a lot of work just to change a variable name, but Ω_r will be defined differently for each of the other types of filter selectivities as we will see in the next sections. For that reason (3.2)–(3.6) do not include the additional subscript *L;* however, in each section we will define Ω_r with a subscript as in (3.1) for clarity.

So in this lowpass case, the first step in the unnormalization procedure doesn't require any work at all. We simply determine the order of the filter as we have in the past. The second step of the unnormalization procedure, determination of the normalized approximation function, has already been developed in the previous chapter. It appears that we are ready to determine the third and final step of the procedure, which is to unnormalize the normalized approximation function. In the lowpass case, this simply requires a scaling of the frequency characteristic from 1 rad/sec to a more usable frequency. A simple substitution for the normalized variable *S* is all that is necessary, as shown in (3.7). (A subscript of *L* is used to indicate that this substitution is for lowpass filters only.) The frequency constant ω_o will be ω_{stop} for the inverse Chebyshev approximation, as discussed in Chapter 2, and ω_{pass} for all other approximations.

$$S_L = \frac{s}{\omega_o} \tag{3.7}$$

3.1.1 Handling a First-Order Factor

We will be developing code to implement the unnormalization process, so it is important to carefully describe the substitution process. For the first-order factor, the process begins with (3.8), where the B_1 coefficient is typically 1:

$$H(s) = \left.\frac{A_1 \cdot S + A_2}{B_1 \cdot S + B_2}\right|_{S=s/\omega_o} = \frac{A_1 \cdot (s/\omega_o) + A_2}{B_1 \cdot (s/\omega_o) + B_2} \tag{3.8}$$

In this equation, uppercase *A* and *B* represent the coefficients of the normalized approximation function. After simplification, (3.9) results in a new set of coefficients. In this equation, lowercase *a* and *b* represent the unnormalized coefficients that will be used in our final approximation function:

$$H(s) = \frac{A_1 \cdot s + A_2 \cdot \omega_o}{B_1 \cdot s + B_2 \cdot \omega_o} = \frac{a_1 \cdot s + a_2}{b_1 \cdot s + b_2} \tag{3.9}$$

We can generalize these results for the first-order factor below:

- The gain constant is unchanged.
- The *s*-term coefficients become

  $a_1 = A_1$, $b_1 = B_1$
- The constant term coefficients become

  $a_2 = A_2\ \omega_o$, $b_2 = B_2\ \omega_o$

3.1.2 Handling a Second-Order Factor

In the case of the quadratic terms that are used to describe our coefficients, the unnormalization process is shown in (3.10) and (3.11):

$$H(s) = \left. \frac{A_0 \cdot S^2 + A_1 \cdot S + A_2}{B_0 \cdot S^2 + B_1 \cdot S + B_2} \right|_{S=s/\omega_o} = \frac{A_0 \cdot (s/\omega_o)^2 + A_1 \cdot (s/\omega_o) + A_2}{B_0 \cdot (s/\omega_o)^2 + B_1 \cdot (s/\omega_o) + B_2} \quad (3.10)$$

$$H(s) = \frac{A_0 \cdot s^2 + A_1 \cdot \omega_o \cdot s + A_2 \cdot \omega_o^2}{B_0 \cdot s^2 + B_1 \cdot \omega_o \cdot s + B_2 \cdot \omega_o^2} = \frac{a_0 \cdot s^2 + a_1 \cdot s + a_2}{b_0 \cdot s^2 + b_1 \cdot s + b_2} \quad (3.11)$$

The coefficient A_0 will be 1 or 0. A value of 1 will be present only if an inverse Chebyshev or elliptic approximation is being unnormalized, while a 0 will be used for Chebyshev and Butterworth. A_1 will normally be 0 for all approximations, but is included for completeness of the derivation in the event we want to use any of our work at a later time when complex conjugate zeros will occur off the $j\omega$ axis. B_0 will typically be 1 for all cases, but is retained for generality. By observation, we can determine the following relationships that can be used in our C code:

- The gain constant is unchanged.
- The s^2-term coefficients become
  $a_0 = A_0$, $b_0 = B_0$
- The s-term coefficients become
  $a_1 = A_1\ \omega_o$, $b_1 = B_1\ \omega_o$
- The constant term coefficients become
  $a_2 = A_2\omega_o^2$, $b_2 = B_2\omega_o^2$

Complete numerical examples of the lowpass unnormalization process are now in order.

Example 3.1 Unnormalized Inverse Chebyshev Lowpass Filter

Problem: Determine the transfer function for an inverse Chebyshev lowpass filter to satisfy the specifications:

$a_{pass} = -0.25$ dB, $a_{stop} = -38.0$ dB,
$\omega_{pass} = 600$ rad/sec, $\omega_{stop} = 1{,}000$ rad/sec

Solution: Using the material of Section 2.4, the important values for this example and the normalized transfer function are listed below. The unnormalized transfer function is then determined by making the substitution $S = s \,/\, \omega_o$ and using the relationships just developed:

$\Omega_r = 1.667$ $n = 5.90$ (6th order) $\omega_o = 1000.0$ rad/sec

$$H_{I,6}(S) = \frac{0.01259 \cdot (S^2 + 1.072) \cdot (S^2 + 2.000) \cdot (S^2 + 14.93)}{(S^2 + 0.2679 \cdot S + 0.5455) \cdot (S^2 + 0.9583 \cdot S + 0.7142) \cdot (S^2 + 1.895 \cdot S + 1.034)}$$

$$H_{I,6}(s) = \frac{0.01259 \cdot (s^2 + 1.072 \cdot 10^6) \cdot (s^2 + 2.000 \cdot 10^6) \cdot (s^2 + 14.93 \cdot 10^6)}{(s^2 + 2{,}679 \cdot s + 545.5 \cdot 10^3) \cdot (s^2 + 958.3 \cdot s + 714.2 \cdot 10^3) \cdot (s^2 + 1{,}895 \cdot s + 1.034 \cdot 10^6)}$$

Example 3.2 Unnormalized Butterworth Lowpass Filter

Problem: Determine the transfer function for a Butterworth lowpass filter to satisfy the following specifications:

$a_{\text{pass}} = -0.5$ dB, $a_{\text{stop}} = -21$ dB,
$f_{\text{pass}} = 1{,}000$ Hz, $f_{\text{stop}} = 2{,}000$ Hz

Solution: Using the material of Section 2.2, the important values for this example and the normalized transfer function are listed below:

$\Omega_r = 2.0$ $n = 5.00$ (5th order) $\omega_o = 6{,}283.19$ rad/sec

$$H_{B,5}(S) = \frac{1.234 \cdot 1.523 \cdot 1.523}{(S + 1.234) \cdot (S^2 + 1.997 \cdot S + 1.523) \cdot (S^2 + 0.7627 \cdot S + 1.523)}$$

The unnormalized transfer function is then determined by making the substitution $S = s \,/\, \omega_o$ and using the relationships just developed:

$$H_{B,5}(s) = \frac{7754 \cdot 6.013 \cdot 10^7 \cdot 6.013 \cdot 10^7}{(s + 7{,}754) \cdot (s^2 + 12.55 \cdot 10^3 \cdot s + 6.013 \cdot 10^7) \cdot (s^2 + 4{,}792 \cdot s + 6.013 \cdot 10^7)}$$

We can also use WFilter to design either of the lowpass filters just described, but in this case we'll pick the Butterworth filter. The coefficients and response of this filter are shown in Figures 3.2 and 3.3.

```
Butterworth Lowpass Filter

Selectivity:        Lowpass
Approximation:      Butterworth
Implementation:     Analog
Passband gain (dB): -0.5
Stopband gain (dB): -21.0
Passband freq (Hz): 1000.0
Stopband freq (Hz): 2000.0

Filter Length/Order: 05
Overall Filter Gain: 1.00000000000E+00

            Numerator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 0.0  0.00000000000E+00  7.75420567954E+03
02 0.0  0.00000000000E+00  6.01277057205E+07
03 0.0  0.00000000000E+00  6.01277057205E+07

           Denominator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 0.0  1.00000000000E+00  7.75420567954E+03
02 1.0  4.79236266571E+03  6.01277057205E+07
03 1.0  1.25465683452E+04  6.01277057205E+07
```

Figure 3.2 Filter coefficients for Example 3.2 from WFilter.

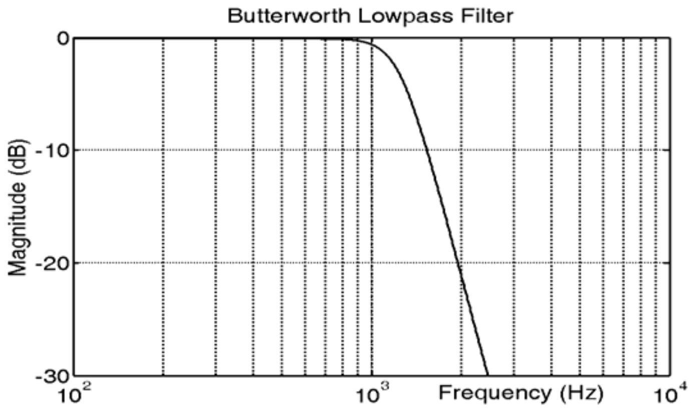


Figure 3.3 Filter magnitude response for Example 3.2.

3.2 UNNORMALIZED HIGHPASS APPROXIMATION FUNCTIONS

The normalized lowpass approximation can also be used to generate the approximation function for a highpass filter. The calculation for the ratio frequency Ω_r is based on the ratio of passband to stopband frequencies, as shown in (3.12). This is the reciprocal of the lowpass case, but since $\omega_{\text{pass}} > \omega_{\text{stop}}$, the result still produces a value greater than 1. The Ω_r ratio always produces a value greater than 1 (as we will see in later sections as well) and as the value gets larger and larger, the order of the filter will reduce as long as other characteristics remain the same.

$$\Omega_{rH} = \frac{\omega_{\text{pass}}}{\omega_{\text{stop}}} = \frac{f_{\text{pass}}}{f_{\text{stop}}} \tag{3.12}$$

Once the normalized lowpass approximation is determined based on the order, we can unnormalize the lowpass transfer function using an appropriate unnormalization substitution. In the case of the highpass filter, the unnormalization substitution is given in (3.13). As in the lowpass case, ω_o will take on the value of ω_{pass} except for the inverse Chebyshev approximation where it will have the value of ω_{stop}.

$$S_H = \frac{\omega_o}{s} \tag{3.13}$$

3.2.1 Handling a First-Order Factor

For the first-order case, we start with (3.14) and make the substitution of (3.13). The final result is then shown in (3.15). In this unnormalization case, we see that there is a gain adjustment (A_2 / B_2) that must be considered.

$$H(s) = \left.\frac{A_1 \cdot S + A_2}{B_1 \cdot S + B_2}\right|_{S=\omega_o/s} = \frac{A_1 \cdot (\omega_o/s) + A_2}{B_1 \cdot (\omega_o/s) + B_2} \tag{3.14}$$

$$H(s) = \frac{A_2}{B_2} \cdot \frac{s + (A_1 / A_2) \cdot \omega_o}{s + (B_1 / B_2) \cdot \omega_o} = \frac{A_2}{B_2} \cdot \frac{a_1 \cdot s + a_2}{b_1 \cdot s + b_2} \tag{3.15}$$

From careful observation we can draw the following information from these equations:

- The gain constant is multiplied by A_2 / B_2.
- The s-term coefficients become
  $a_1 = 1$, $b_1 = 1$
- The constant term coefficients become
  $a_2 = (A_1 / A_2)\ \omega_o$, $b_2 = (B_1 / B_2)\ \omega_o$

3.2.2 Handling a Second-Order Factor

In the case of this highpass unnormalization process, the second-order factors will be unnormalized in the manner shown in (3.16):

$$H(s) = \left.\frac{A_0 \cdot S^2 + A_1 \cdot S + A_2}{B_0 \cdot S^2 + B_1 \cdot S + B_2}\right|_{S=\omega_o/s} = \frac{A_0 \cdot (\omega_o/s)^2 + A_1 \cdot (\omega_o/s) + A_2}{B_0 \cdot (\omega_o/s)^2 + B_1 \cdot (\omega_o/s) + B_2} \qquad (3.16)$$

that can be simplified to produce

$$H(s) = \frac{A_2}{B_2} \cdot \frac{s^2 + (A_1/A_2) \cdot \omega_o \cdot s + (A_0/A_2) \cdot \omega_o^2}{s^2 + (B_1/B_2) \cdot \omega_o \cdot s + (B_0/B_2) \cdot \omega_o^2} = \frac{A_2}{B_2} \cdot \frac{a_0 \cdot s^2 + a_1 \cdot s + a_2}{b_0 \cdot s^2 + b_1 \cdot s + b_2} \qquad (3.17)$$

Notice that if A_0 and A_1 are both zero (which will be the case for Butterworth and Chebyshev approximations), then a_1 and a_2 will be zero, leaving only an s^2-term in the numerator. A_2 will never be zero in a normalized approximation function. We can summarize the results of the unnormalization below:

- The gain constant is multiplied by A_2 / B_2.
- The s^2-term coefficients become
  $a_0 = 1$, $b_0 = 1$
- The s-term coefficients become
  $a_1 = (A_1 / A_2)\ \omega_o$, $b_1 = (B_1 / B_2)\ \omega_o$
- The constant term coefficients become
  $a_2 = (A_0 / A_2)\omega_o^2,\ b_2 = (B_0 / B_2)\omega_o^2$

Numerical examples of the highpass unnormalization process can now be used to better illustrate the process.

Example 3.3 Unnormalized Elliptic Highpass Filter

Problem: Determine the transfer function for an elliptic highpass filter to satisfy the following specifications:

$a_{\text{pass}} = -0.5$ dB, $a_{\text{stop}} = -45.0$ dB,
$\omega_{\text{pass}} = 3{,}000$ rad/sec, $\omega_{\text{stop}} = 2{,}000$ rad/sec

Solution: Using the material of Section 2.5, the important values for this example and the normalized transfer function are listed below:

$\Omega_r = 1.5$ $n = 4.61$ (5th order) $\omega_o = 3{,}000.0$ rad/sec

$$H_{E,5}(S) = \frac{0.04507 \cdot 0.4260 \cdot (S^2 + 5.438) \cdot (S^2 + 2.426)}{(S + 0.4260) \cdot (S^2 + 0.5702 \cdot S + 0.5760) \cdot (S^2 + 0.1635 \cdot S + 1.032)}$$

The unnormalized transfer function is then determined by making the substitution $S = \omega_o / s$ and using the relationships just developed:

$$H_{E,5}(s) = \frac{s \cdot (s^2 + 1.655 \cdot 10^6) \cdot (s^2 + 3.711 \cdot 10^6)}{(s + 7{,}043) \cdot (s^2 + 2{,}970 \cdot s + 1.562 \cdot 10^7) \cdot (s^2 + 475.2 \cdot s + 8.722 \cdot 10^6)}$$

Example 3.4 Unnormalized Chebyshev Highpass Filter

Problem: Determine the transfer function for a Chebyshev highpass filter to satisfy the following specifications:

$a_{\text{pass}} = -1.5$ dB, $a_{\text{stop}} = -40$ dB,
$f_{\text{pass}} = 2{,}000$ Hz, $f_{\text{stop}} = 800$ Hz

Solution: Using the material of Section 2.3, the important values for this example and the normalized transfer function are listed below:

$\Omega_r = 2.5$ $n = 3.66$ (4th order) $\omega_o = 12{,}566.4$ rad/sec

$$H_{C,4}(S) = \frac{0.8414 \cdot 0.9505 \cdot 0.2434}{(S^2 + 0.2383 \cdot S + 0.9505) \cdot (S^2 + 0.5752 \cdot S + 0.2434)}$$

The unnormalized transfer function is then determined by making the substitution $S = \omega_o / s$ and using the relationships just developed:

$$H_{C,4}(s) = \frac{0.8414 \cdot s^2 \cdot s^2}{(s^2 + 3{,}150 \cdot s + 1.661 \cdot 10^8) \cdot (s^2 + 29{,}703 \cdot s + 6.489 \cdot 10^8)}$$

The results of using WFilter to design the filter of Example 3.4 are illustrated in Figures 3.4 and 3.5, which show the coefficients and magnitude response.

3.3 UNNORMALIZED BANDPASS APPROXIMATION FUNCTIONS

In the case of a bandpass unnormalization, Ω_r will be defined in (3.18). Note that for this case, Ω_r will be greater than 1, as has been the case for the lowpass and highpass unnormalization.

```
Chebyshev Highpass Filter

Selectivity:         Highpass
Approximation:       Chebyshev
Implementation:      Analog
Passband gain (dB):  -1.5
Stopband gain (dB):  -40.0
Passband freq (Hz):  2000.0
Stopband freq (Hz):  800.0

Filter Length/Order: 04
Overall Filter Gain: 8.41395141645E-01

            Numerator Coefficients
QD [S^2 +       S          +        1       ]
== ==========================================
01 1.0  0.00000000000E+00  0.00000000000E+00
02 1.0  0.00000000000E+00  0.00000000000E+00

           Denominator Coefficients
QD [S^2 +       S          +        1       ]
== ==========================================
01 1.0  3.15012807725E+03  1.66143895400E+08
02 1.0  2.97027255443E+04  6.48898535622E+08
```

Figure 3.4 Filter coefficients for Example 3.4 from WFilter.

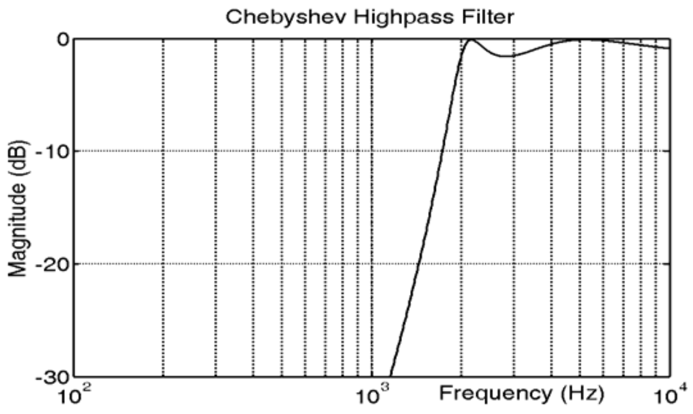


Figure 3.5 Filter magnitude response for Example 3.4.

$$\Omega_{rP} = \frac{\omega_{\text{stop2}} - \omega_{\text{stop1}}}{\omega_{\text{pass2}} - \omega_{\text{pass1}}} = \frac{f_{\text{stop2}} - f_{\text{stop1}}}{f_{\text{pass2}} - f_{\text{pass1}}} \tag{3.18}$$

After determining the normalized lowpass approximation using the order we determined from the bandpass specifications, we can unnormalize the lowpass function into a bandpass function. To accomplish this we use the substitution given in (3.19):

$$S_P = \frac{s^2 + \omega_o^2}{BW \cdot s} \tag{3.19}$$

where for all approximations except the inverse Chebyshev,

$$\omega_o = \sqrt{\omega_{\text{pass1}} \cdot \omega_{\text{pass2}}} \tag{3.20}$$

$$BW = \omega_{\text{pass2}} - \omega_{\text{pass1}} \tag{3.21}$$

For the inverse Chebyshev case these values are defined as (another example of the opposite nature of this approximation)

$$\omega_o = \sqrt{\omega_{\text{stop1}} \cdot \omega_{\text{stop2}}} \tag{3.22}$$

$$BW = \omega_{\text{stop2}} - \omega_{\text{stop1}} \tag{3.23}$$

In order to provide an accurate value of Ω_r in (3.18), the stopband and passband edge frequencies must be symmetrically spaced on either side of ω_o. The simplest way to test for this is to check to see that the relationship of (3.24) is satisfied. If this equation is not satisfied, the larger side must be reduced to form an equality by increasing ω_{stop1} or decreasing ω_{stop2}. This will tighten the restrictions, so the original specifications will still be met, and an accurate order can be calculated. (If there is an extreme inequality, other measures can be used to implement the filter. For example, an additional lowpass or highpass filter can be added to provide the required selectivity.)

$$\frac{\omega_{\text{pass1}}}{\omega_{\text{stop1}}} = \frac{\omega_{\text{stop2}}}{\omega_{\text{pass2}}} \tag{3.24}$$

As indicated by (3.19), the unnormalization process will result in a bandpass approximation function that has twice the order of the lowpass function used to generate it. This seems reasonable when we consider that a bandpass filter must provide a transition from a stopband to a passband (like a highpass filter) and another transition from a passband to a stopband (like a lowpass filter). The resulting function must therefore be twice the order of the original lowpass function on which it is based.

3.3.1 Handling a First-Order Factor

For a first-order factor in the lowpass approximation function, (3.25) shows how the substitution of (3.19) is made:

$$H(s) = \left.\frac{A_1 \cdot S + A_2}{B_1 \cdot S + B_2}\right|_{S=(s^2+\omega_o^2)/(BW\cdot s)} = \frac{A_1 \cdot [(s^2+\omega_o^2)/(BW\cdot s)] + A_2}{B_1 \cdot [(s^2+\omega_o^2)/(BW\cdot s)] + B_2} \quad (3.25)$$

And after some simplification we have the result in (3.26). The relationships between the coefficients are shown. Note that if $A_1 = 0$, as will normally be the case, the numerator will only have an s-term present.

$$H(s) = \frac{A_1 \cdot s^2 + A_2 \cdot BW \cdot s + A_1 \cdot \omega_o^2}{B_1 \cdot s^2 + B_2 \cdot BW \cdot s + B_1 \cdot \omega_o^2} = \frac{a_0 \cdot s^2 + a_1 \cdot s + a_2}{b_0 \cdot s^2 + b_1 \cdot s + b_2} \quad (3.26)$$

- The gain constant is unchanged.
- The s^2-term bandpass coefficients become
  $a_0 = A_1$, $b_0 = B_1$
- The s-term bandpass coefficients become
  $a_1 = A_2\, BW$, $b_1 = B_2\, BW$
- The constant term bandpass coefficients become
  $a_2 = A_1\omega_o^2$, $b_2 = B_1\omega_o^2$

3.3.2 Handling a Second-Order Factor

Unnormalizing a second-order factor is a bit more of a challenge. When the substitution variable S_P of (3.19) is inserted into a second-order lowpass approximation, a fourth-order factor results. What do we do with a fourth-order factor? All of our development to this point is based on quadratic factors and with good reason. They represent a complex conjugate pair and they will be used to efficiently implement the filters in later chapters. We could factor the fourth-order, but this would require a numerical algorithm that is time-consuming and not always accurate. There is another directed procedure that can be used.

If we factor the lowpass approximation quadratic into two complex conjugate factors before making the substitution of (3.19), the result after the substitution and simplification is two quadratic equations. However, each of these quadratics would have a complex coefficient that would mean they could not be implemented directly. However, if these two quadratics are again factored, we will find two sets of complex conjugate pairs within the set of four factors. These complex conjugate pairs could then be combined to produce two quadratics that have all real coefficients.

Perhaps an easy example is in order. Consider the transfer function shown in (3.27) that has already been factored:

$$H(S) = \frac{2}{S^2 + 2 \cdot S + 2} = \frac{2}{(S + 1 + j1) \cdot (S + 1 - j1)} \tag{3.27}$$

Now if we assume that $\omega_o = 1$ and $BW = 1$, we can substitute $S = (s^2 + 1) / s$ and simplify to produce the following:

$$H(s) = \frac{2 \cdot s^2}{[s^2 + (1 + j1) \cdot s + 1] \cdot [s^2 + (1 - j1) \cdot s + 1]} \tag{3.28}$$

The roots of the first quadratic can be determined to be

$$s_{1,2} = \frac{-(1 + j1) \pm (0.486 + j2.058)}{2} = (-0.257 + j0.529), (-0.743 - j1.53) \tag{3.29a}$$

and the two roots of the second quadratic pair up with the first.

$$s_{3,4} = (-0.257 - j0.529), (-0.743 + j1.53) \tag{3.29b}$$

The resulting transfer function can then be written as (3.30) by combining the complex conjugate roots from each quadratic:

$$H(s) = \frac{2 \cdot s^2}{(s^2 + 0.514 \cdot s + 0.346) \cdot (s^2 + 1.486 \cdot s + 2.890)} \tag{3.30}$$

This algorithm for finding the two quadratics in the bandpass approximation from the single quadratic in the lowpass function will be used as the standard method in this section. Unfortunately, it is very difficult to define the final bandpass coefficients in terms of only the initial lowpass coefficients because of the complexity of expressions. However, if we use a few intermediate variables, the process should be able to be demonstrated without too much confusion.

We start with the general expression for the normalized lowpass second-order factor shown in (3.31) in normal and factored form. The A_0 and B_0 coefficients have been omitted for clarity since they will be assumed to be 1 in this case, and p and z represent the complex poles and zeros, respectively. (An asterisk indicates the complex conjugate value.)

$$H(s) = \frac{S^2 + A_1 \cdot S + A_2}{S^2 + B_1 \cdot S + B_2} = \left.\frac{(S + z_1)(S + z_1^*)}{(S + p_1)(S + p_1^*)}\right|_{S=(s^2+\omega_o^2)/(BW\cdot s)} \tag{3.31}$$

After making the indicated substitution and simplifying, we have

$$H(s) = \frac{(s^2 + BW \cdot z_1 \cdot s + \omega_o^2) \cdot (s^2 + BW \cdot z_1^* \cdot s + \omega_o^2)}{(s^2 + BW \cdot p_1 \cdot s + \omega_o^2) \cdot (s^2 + BW \cdot p_1^* \cdot s + \omega_o^2)} \tag{3.32}$$

Each one of the quadratic factors in (3.32) can now be factored into first-order factors, as indicated in (3.33). Note that the constants from the z_1 quadratic are labeled with an *a* and *b*, while the complex conjugates use *c* and *d*. The denominator uses the same designations.

$$H(s) = \frac{(s + z_{1a}) \cdot (s + z_{1b}) \cdot (s + z_{1c}) \cdot (s + z_{1d})}{(s + p_{1a}) \cdot (s + p_{1b}) \cdot (s + p_{1c}) \cdot (s + p_{1d})} \tag{3.33}$$

Now, by matching the complex conjugate pairs, we can reconstruct two quadratics with real coefficients in both the numerator and denominator:

$$H(s) = \frac{(s^2 + a_1 \cdot s + a_2)(s^2 + a_4 \cdot s + a_5)}{(s^2 + b_1 \cdot s + b_2)(s^2 + b_4 \cdot s + b_5)} \tag{3.34}$$

The results shown in (3.34) are valid for the inverse Chebyshev and elliptic approximation functions that use zeros on the $j\omega$ axis. However, for the Butterworth and Chebyshev approximation functions, the result will be somewhat different. Equation (3.35) shows the starting point for this development. After substitution and simplification, (3.36) results. Then following the same basic steps as in the previous derivation, (3.37) eventually emerges.

$$H(s) = \frac{A_2}{S^2 + B_1 \cdot S + B_2} = \left. \frac{A_2}{(S + p_1)(S + p_1^*)} \right|_{S=(s^2+\omega_o^2)/(BW \cdot s)} \tag{3.35}$$

$$H(s) = \frac{A_2 \cdot BW^2 \cdot s^2}{(s^2 + BW \cdot p_1 \cdot s + \omega_o^2) \cdot (s^2 + BW \cdot p_1^* \cdot s + \omega_o^2)} \tag{3.36}$$

$$H(s) = \frac{A_2 \cdot BW^2 \cdot s^2}{(s^2 + b_1 \cdot s + b_2)(s^2 + b_4 \cdot s + b_5)} \tag{3.37}$$

Now, some examples illustrating the bandpass design process are in order.

Example 3.5 Unnormalized Butterworth Bandpass Filter

Problem: Determine the transfer function for a Butterworth bandpass filter to satisfy the following specifications:

$a_{\text{pass}} = -1.0$ dB,	$a_{\text{stop}} = -21$ dB,	$f_{\text{pass1}} = 300$ Hz,
$f_{\text{pass2}} = 3{,}000$ Hz,	$f_{\text{stop1}} = 50$ Hz,	$f_{\text{stop2}} = 9{,}000$ Hz

Solution: Using the material of Section 2.2, the important values for this example and the normalized transfer function are listed below. In this case, f_{stop1} must be changed to 100 Hz to provide symmetry. The function is shown with quadratics in factored form, as indicated by the (2) superscript.

$\Omega_r = 3.3$	$n = 2.59$ (3rd order)
$\omega_o = 5{,}960.8$ rad/sec	$BW = 16{,}965$ rad/sec

$$H_{B,3}(S) = \frac{1.2526 \cdot 1.5690}{(S + 1.2526) \cdot (S + 0.62629 \pm j1.0848)^{(2)}}$$

After making the substitution of (3.19) and factoring again, the following equation emerges:

$$H_{B,6}(s) = \frac{1.253 \cdot 1.569 \cdot (16{,}965 \cdot s)^3}{(s^2 + 21{,}249 \cdot s + 3.553 \cdot 10^7) \cdot (s + 716.1 \pm j1434)^{(2)} \cdot (s + 9909 \pm j19{,}836)^{(2)}}$$

After simplification, the following transfer function results:

$$H_{B,6}(s) = \frac{(2.125\cdot 10^4)^3 \cdot s^3}{(s^2+21{,}249s+3.553\cdot 10^7)\cdot(s^2+1{,}432s+2.568\cdot 10^6)\cdot(s^2+19{,}817s+4.917\cdot 10^8)}$$

Example 3.6 Unnormalized Inverse Chebyshev Bandpass Filter

Problem: Determine the transfer function for an inverse Chebyshev bandpass filter to satisfy the following specifications:

a_{pass} = –0.5 dB, a_{stop} = –33 dB, f_{pass1} = 100 Hz,
f_{pass2} = 200 Hz, f_{stop1} = 50 Hz, f_{stop2} = 400 Hz

Solution: Using the material of Section 2.4, the important values for this example and the normalized transfer function are listed below. The transfer function is shown with quadratics in factored form, as indicated by the (2) superscript.

Ω_r = 3.5 n = 2.88 (3rd order)

ω_o = 888.58 rad/sec BW = 2,199.1 rad/sec

$$H_{I,3}(S) = \frac{0.14264\cdot 0.47098\cdot(S\pm j1.15470)^{(2)}}{(S+0.47098)\cdot(S+0.20190\pm j0.38655)^{(2)}}$$

After making the substitution of (3.19) and factoring again, the following equation emerges:

$$H_{I,6}(s) = \frac{0.14264\cdot(1{,}036\cdot s)\cdot(s\pm j280.1)^{(2)}\cdot(s\pm j2{,}819)^{(2)}}{(s^2+1{,}036\cdot s+7.896\cdot 10^5)\cdot(s+124.2\pm j539.6)^{(2)}\cdot(s+319.8\pm j1{,}390)^{(2)}}$$

After simplification, the following transfer function results:

$$H_{I,6}(s) = \frac{0.14264\cdot(1{,}036\cdot s)\cdot(s^2+7.843\cdot 10^4)\cdot(s^2+7.949\cdot 10^6)}{(s^2+1{,}036\cdot s+7.896\cdot 10^5)\cdot(s^2+248.4\cdot s+3.066\cdot 10^5)\cdot(s^2+639.6\cdot s+2.033\cdot 10^6)}$$

We can use WFilter to design the inverse Chebyshev bandpass filter of Example 3.6. The coefficients for this design are shown in Figure 3.6 with the magnitude response shown in Figure 3.7. Notice that two of the numerator coefficients are quite small (approximately 10^{-16} or smaller) and should be interpreted as zero since these quadratics represent complex zeros located on the $j\omega$-axis.

```
Inverse Chebyshev Bandpass Filter

Selectivity:         Bandpass
Approximation:       Inv. Chebyshev
Implementation:      Analog
Passband gain (dB): -0.5
Stopband gain (dB): -33.0
PB freq-lower (Hz): 100.0
PB freq-upper (Hz): 200.0
SB freq-lower (Hz): 50.0
SB freq-upper (Hz): 400.0

Filter Length/Order: 06
Overall Filter Gain: 1.42635925362E-01

            Numerator Coefficients
QD [S^2 +        S         +        1       ]
== ==========================================
01 0.0  1.03573607270E+03  0.00000000000E+00
02 1.0  1.52838376237E-16  7.84287312832E+04
03 1.0 -1.51816114564E-17  7.94884951494E+06

           Denominator Coefficients
QD [S^2 +        S         +        1       ]
== ==========================================
01 1.0  1.03573607270E+03  7.89568352087E+05
02 1.0  2.48367370656E+02  3.06585558034E+05
03 1.0  6.39635528883E+02  2.03342318737E+06
```

Figure 3.6 Filter coefficients for Example 3.6 from WFilter.

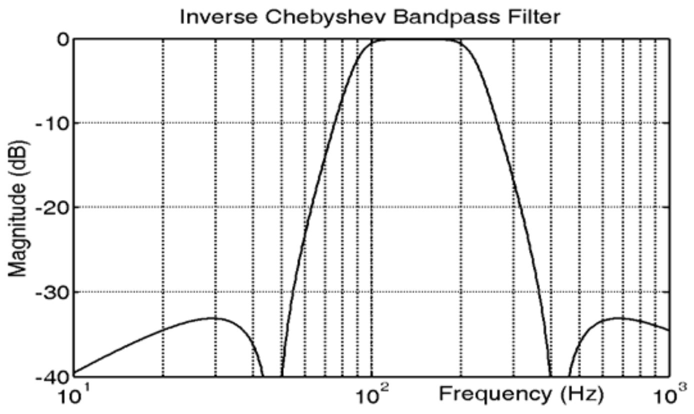


Figure 3.7 Filter magnitude response for Example 3.6.

3.4 UNNORMALIZED BANDSTOP APPROXIMATION FUNCTIONS

We will find that the unnormalization of the lowpass normalized function into a bandstop approximation is very similar to the bandpass case. The first step in the procedure is to determine the order required from the lowpass approximation based on the bandstop specifications. The value of Ω_r to use in the bandstop case is shown in (3.38), which is the reciprocal of the bandpass case. We notice again that Ω_r will be greater than 1:

$$\Omega_r = \frac{\omega_{\text{pass2}} - \omega_{\text{pass1}}}{\omega_{\text{stop2}} - \omega_{\text{stop1}}} = \frac{f_{\text{pass2}} - f_{\text{pass1}}}{f_{\text{stop2}} - f_{\text{stop1}}} \tag{3.38}$$

As in the bandpass case, (3.24) must be satisfied in order to get an accurate value of Ω_r, except in this case either ω_{pass1} must be increased or ω_{pass2} must be decreased to achieve equality. After finding the order, we can unnormalize the lowpass function into a bandstop function using the substitution given in (3.39):

$$S_S = \frac{BW \cdot s}{s^2 + \omega_o^2} \tag{3.39}$$

As before, all approximations except the inverse Chebyshev will define

$$\omega_o = \sqrt{\omega_{\text{pass1}} \cdot \omega_{\text{pass2}}} \tag{3.40}$$

$$BW = \omega_{\text{pass2}} - \omega_{\text{pass1}} \tag{3.41}$$

while for the inverse Chebyshev case

$$\omega_o = \sqrt{\omega_{\text{stop1}} \cdot \omega_{\text{stop2}}} \tag{3.42}$$

$$BW = \omega_{\text{stop2}} - \omega_{\text{stop1}} \tag{3.43}$$

The resultant bandstop approximation function will be twice the order of the normalized lowpass function just as in the bandpass case. The bandstop filter is in effect implementing both a lowpass and highpass filter and therefore requires twice the order.

3.4.1 Handling a First-Order Factor

Equation (3.44) shows how a first-order factor is unnormalized into a second-order factor using the substitution of (3.39):

$$H(S) = \left.\frac{A_1 \cdot S + A_2}{B_1 \cdot S + B_2}\right|_{S=(BW\cdot s)/(s^2+\omega_o^2)} = \frac{A_1 \cdot [(BW \cdot s)/(s^2+\omega_o^2)] + A_2}{B_1 \cdot [(BW \cdot s)/(s^2+\omega_o^2)] + B_2} \qquad (3.44)$$

Equation (3.45) shows the result after simplification followed by the observations that can be made for this case:

$$H(s) = \frac{A_2}{B_2} \cdot \frac{s^2 + (A_1/A_2) \cdot BW \cdot s + \omega_o^2}{s^2 + (B_1/B_2) \cdot BW \cdot s + \omega_o^2} = \frac{a_0 \cdot s^2 + a_1 \cdot s + a_2}{b_0 \cdot s^2 + b_1 \cdot s + b_2} \qquad (3.45)$$

- The gain constant is multiplied by A_2 / B_2.
- The s^2-term bandstop coefficients become
  $a_0 = 1, b_0 = 1$
- The s-term bandstop coefficients become
  $a_1 = (A_1 / A_2)\, BW, b_1 = (B_1 / B_2)\, BW$
- Constant term bandstop coefficients become
  $a_2 = \omega_o^2,\ b_2 = \omega_o^2$

3.4.2 Handling a Second-Order Factor

In order to unnormalize a second-order factor we experience the same problems as in the bandpass case. A direct substitution would give us a fourth-order transfer function, which is not what we want. However, the methodology used in the bandpass case does work in this case as well. The procedure is outlined below for the bandstop case that has a few differences due to a different substitution factor.

Starting at the same point as with the bandpass case, (3.46) shows the result of the factoring of the initial quadratics:

$$H(s) = \frac{S^2 + A_1 \cdot S + A_2}{S^2 + B_1 \cdot S + B_2} = \left.\frac{(S+z_1)(S+z_1^*)}{(S+p_1)(S+p_1^*)}\right|_{S=(BW\cdot s)/(s^2+\omega_o^2)} \qquad (3.46)$$

Equation (3.47) results after the substitution and simplification:

$$H(s) = \frac{z_1 \cdot z_1^*}{p_1 \cdot p_1^*} \cdot \frac{[s^2 + (BW/z_1) \cdot s + \omega_o^2] \cdot [s^2 + (BW/z_1^*) \cdot s + \omega_o^2]}{[s^2 + (BW/p_1) \cdot s + \omega_o^2] \cdot [s^2 + (BW/p_1^*) \cdot s + \omega_o^2]} \qquad (3.47)$$

The initial gain factor of (3.47) can be shown to be A_2 / B_2. The quadratic factors can be factored into first-order factors, as indicated in (3.48):

$$H(s) = \frac{A_2}{B_2} \cdot \frac{(s+z_{1a}) \cdot (s+z_{1b}) \cdot (s+z_{1c}) \cdot (s+z_{1d})}{(s+p_{1a}) \cdot (s+p_{1b}) \cdot (s+p_{1c}) \cdot (s+p_{1d})} \tag{3.48}$$

We then reconstruct two quadratics in the numerator and denominator by matching the complex conjugate pairs:

$$H(s) = \frac{A_2}{B_2} \cdot \frac{(s^2 + a_1 \cdot s + a_2)(s^2 + a_4 \cdot s + a_5)}{(s^2 + b_1 \cdot s + b_2)(s^2 + b_4 \cdot s + b_5)} \tag{3.49}$$

This result is valid for the rational approximation functions (inverse Chebyshev and elliptic), but for the all-pole approximations (Butterworth and Chebyshev), we must develop a slightly different version. Equations (3.50) and (3.51) show the factoring and substitution of (3.39). After the quadratics of (3.51) are factored and the matching complex conjugate terms are combined, the final form is (3.52) .

$$H(s) = \frac{A_2}{S^2 + B_1 \cdot S + B_2} = \left.\frac{A_2}{(S + p_1)(S + p_1^*)}\right|_{S=(BW \cdot s)/(s^2+\omega^2)} \tag{3.50}$$

$$H(s) = \frac{A_2}{p_1 \cdot p_1^*} \cdot \frac{(s^2 + \omega_o^2)^2}{[s^2 + (BW/p_1) \cdot s + \omega_o^2] \cdot [s^2 + (BW/p_1^*) \cdot s + \omega_o^2]} \tag{3.51}$$

$$H(s) = \frac{A_2}{B_2} \cdot \frac{(s^2 + \omega_o^2)^2}{(s^2 + b_1 \cdot s + b_2)(s^2 + b_4 \cdot s + b_5)} \tag{3.52}$$

Complete numerical examples of the bandstop unnormalization process are given next.

Example 3.7 Unnormalized Chebyshev Bandstop Filter

Problem: Determine the transfer function for a Chebyshev bandstop filter to satisfy the following specifications:

ω_{pass1} = 3,000 rad/sec, ω_{pass2} = 24,000 rad/sec, ω_{stop1} = 6,000 rad/sec,
ω_{stop2} = 12,000 rad/sec, a_{pass} = –1.0 dB, a_{stop} = –35 dB

Solution: Using the material of Section 2.3, the important values for this example and the normalized transfer function are listed below. The transfer function is shown with quadratics in factored form as shown by the (2) superscript.

$\Omega_r = 3.5$ $n = 2.80$ (3rd order)

$\omega_o = 8485.3$ rad/sec $BW = 21{,}000$ rad/sec

$$H_{C,3}(S) = \frac{0.49417 \cdot 0.99421}{(S + 0.49417) \cdot (S + 0.24709 \pm j0.96600)^{(2)}}$$

After making the substitution of (3.39) and factoring again, the following equation emerges, which can be simplified into the final result:

$$H_{C,6}(s) = \frac{(s^2 + 7.200 \cdot 10^7)^3}{(s^2 + 42{,}500 \cdot s + 7.200 \cdot 10^7) \cdot (s + 587.6 \pm j2{,}965)^{(2)} \cdot (s + 4{,}632 \pm j23{,}370)^{(2)}}$$

$$H_{C,6}(s) = \frac{(s^2 + 7.200 \cdot 10^7)^3}{(s^2 + 42{,}495 \cdot s + 7.200 \cdot 10^7)(s^2 + 1{,}175 \cdot s + 9.134 \cdot 10^6)(s^2 + 9{,}263 \cdot s + 5.676 \cdot 10^8)}$$

Example 3.8 Unnormalized Elliptic Bandstop Filter

Problem: Determine the transfer function for an elliptic bandstop filter to satisfy the following specifications:

a_{pass} = –0.3 dB, a_{stop} = –50 dB, f_{pass1} = 50 Hz,
f_{pass2} = 72 Hz, f_{stop1} = 58 Hz, f_{stop2} = 62 Hz

Solution: Using the material of Section 2.5, the important values for this example and the normalized transfer function are listed below. In this case, f_{pass2}

must be changed to 71.92 Hz to provide symmetry. The transfer function is shown with quadratics in factored form as shown by the (2) superscript.

$\Omega_r = 3.3$ $n = 2.75$ (3rd order)

$\omega_o = 376.78$ rad/sec $BW = 137.73$ rad/sec

$$H_{E,3}(S) = \frac{0.73880 \cdot (S \pm j6.31445)}{(S + 0.73880) \cdot (S + 0.35753 \pm j0.93390)}$$

After making the substitution of (3.39) and factoring again, the following equation emerges:

$$H_{I,6}(s) = \frac{(s^2 + 141{,}960) \cdot (s \pm j387.9)^{(2)} \cdot (s \pm j366.0)^{(2)}}{(s^2 + 186.4 \cdot s + 141{,}960) \cdot (s + 16.30 \pm j323.0)^{(2)} \cdot (s + 22.13 \pm j438.4)^{(2)}}$$

After simplification, the following transfer function results:

$$H_{E,6}(s) = \frac{(s^2 + 141{,}964) \cdot (s^2 + 150{,}424) \cdot (s^2 + 133{,}981)}{(s^2 + 186.4 \cdot s + 141{,}964) \cdot (s^2 + 32.60 \cdot s + 104{,}585) \cdot (s^2 + 44.25 \cdot s + 192{,}704)}$$

We can also use WFilter to design the bandstop elliptic filter of Example 3.8. The results of this filter design are shown in Figures 3.8 and 3.9 that show the coefficients and magnitude response, respectively. Again, as in the bandpass case, a couple of the numerator coefficients are very small and can be considered zero.

3.5 ANALOG FREQUENCY RESPONSE

Up to this point we have developed the necessary foundation to design a variety of analog filters. We have calculated the coefficients and are ready to implement the filter in hardware. But before we address the implementation issues in the next chapter, we need to check our design by determining the frequency response of the filter and comparing it to our design specifications. We will discuss the calculation of the frequency response of our filters and also view the C code for the frequency response calculation.

3.5.1 Mathematics for Frequency Response Calculation

The filter approximation function, which we have just determined by the calculation of the unnormalized coefficients, represents a transfer function of a linear system in the *s*-domain. In order to determine the frequency response of the transfer function, we must substitute $j\omega$ for each of the *s*-variables in that transfer function. For example, (3.53) shows a transfer function with one quadratic factor, while (3.54) shows the frequency response for that transfer function:

```
Elliptic Bandstop Filter

Selectivity:        Bandstop
Approximation:      Elliptic
Implementation:     Analog
Passband gain (dB): -0.3
Stopband gain (dB): -50.0
PB freq-lower (Hz): 50.0
PB freq-upper (Hz): 71.92
SB freq-lower (Hz): 58.0
SB freq-upper (Hz): 62.0

Filter Length/Order: 06
Overall Filter Gain: 1.00000000000E+00

            Numerator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 1.0  0.00000000000E+00  1.41964389705E+05
02 1.0  2.10251277992E-17  1.33980657885E+05
03 1.0 -1.98427257744E-17  1.50423861642E+05

           Denominator Coefficients
QD [S^2 +       S         +        1       ]
== ========================================
01 1.0  1.86421021332E+02  1.41964389705E+05
02 1.0  3.26004384421E+01  1.04584515861E+05
03 1.0  4.42522615272E+01  1.92704319359E+05
```

Figure 3.8 Filter coefficients for Example 3.8 from WFilter.

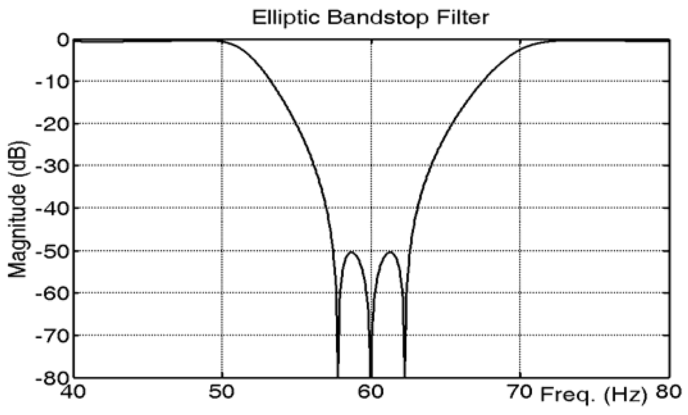


Figure 3.9 Filter magnitude response for Example 3.8.

$$H(s) = \frac{a_o \cdot s^2 + a_1 \cdot s + a_2}{b_o \cdot s^2 + b_1 \cdot s + b_2} \tag{3.53}$$

$$H(s)\big|_{s=j\omega} = H(j\omega) = \frac{a_o \cdot (j\omega)^2 + a_1 \cdot (j\omega) + a_2}{b_o \cdot (j\omega)^2 + b_1 \cdot (j\omega) + b_2} \tag{3.54}$$

After simplification, the frequency response $H(j\omega)$ is shown as a frequency dependent complex number in (3.55). This complex number can be represented in either rectangular form or polar form. However, when we deal with a frequency response, the polar form is the more natural form because the standard frequency response is composed of both a magnitude and phase response portion. Equation (3.56) shows the result of converting (3.55) into polar form:

$$H(j\omega) = \frac{(a_2 - a_o \cdot \omega^2) + j(a_1 \cdot \omega)}{(b_2 - b_o \cdot \omega^2) + j(b_1 \cdot \omega)} \tag{3.55}$$

$$H(j\omega) = \frac{M_a \angle \tan^{-1}(P_a)}{M_b \angle \tan^{-1}(P_b)} \tag{3.56}$$

where

$$M_a = \sqrt{(a_2 - a_o\omega^2)^2 + (a_1\omega)^2}$$

$$M_b = \sqrt{(b_2 - b_o\omega^2)^2 + (b_1\omega)^2}$$

$$P_a = (a_1\omega)/(a_2 - a_o\omega^2)$$

$$P_b = (b_1\omega)/(b_2 - b_o\omega^2)$$

Of course, if the original transfer function has multiple quadratic terms, as our approximation functions do, the total frequency response is dependent on all of the quadratics. The total magnitude result will be the product of the individual magnitudes and the total phase result will be the sum of the individual phases as shown in (3.57), where q represents the number of quadratic factors:

$$H_t(j\omega) = \frac{\prod_{a=1}^{q} M_a \angle \sum_{a=1}^{q} \tan^{-1}(P_a)}{\prod_{b=1}^{q} M_b \angle \sum_{b=1}^{q} \tan^{-1}(P_b)} \tag{3.57}$$

The total frequency response $H_t(j\omega)$ can then be described as shown in (3.58), where the total magnitude is the numerator magnitude divided by the denominator magnitude, and the total phase angle is the denominator phase subtracted from the numerator phase:

$$H_t(j\omega) = M_t \angle \Phi_t \tag{3.58}$$

where

$$M_t = \prod_{a=1}^{q} M_a \Big/ \prod_{b=1}^{q} M_b$$

$$\Phi_t = \sum_{a=1}^{q} \tan^{-1}(P_a) - \sum_{b=1}^{q} \tan^{-1}(P_b)$$

Example 3.9 Frequency Response of a Highpass Filter

Problem: Determine the frequency response (both magnitude and phase) at the passband edge frequency of 2,000 Hz and the stopband edge frequency of 800 Hz for the Chebyshev highpass filter designed in Example 3.4. Determine if the gain specifications of a_{pass} = –1.5 dB and a_{stop} = –40 dB are met. The transfer function is shown below:

$$H_{C,4}(s) = \frac{0.84140 \cdot s^2 \cdot s^2}{(s^2 + 3{,}150.1 \cdot s + 1.6614 \cdot 10^8) \cdot (s^2 + 29{,}703 \cdot s + 6.4890 \cdot 10^8)}$$

Solution: We first make the substitution of $s = j\omega$ into the transfer function to obtain the frequency response $H(j\omega)$. Then after collecting the real and imaginary terms and arranging them, the following equation for the frequency response results:

$$H_{C,4}(j\omega) = \frac{0.8414 \cdot \omega^4}{[(1.6614 \cdot 10^8 - \omega^2) + j3{,}150.1 \cdot \omega] \cdot [(6.4890 \cdot 10^8 - \omega^2) + j29{,}703 \cdot \omega]}$$

This frequency response equation can first be used to determine the frequency response at the stopband frequency of 800 Hz.

$$H_{C,4}(j1{,}600\pi) = \frac{5.3713 \cdot 10^{14}}{(1.4087 \cdot 10^{8} + j1.5834 \cdot 10^{7}) \cdot (6.2363 \cdot 10^{8} + j1.4930 \cdot 10^{8})}$$

$$H_{C,4}(j1{,}600\pi) = \frac{5.3713 \cdot 10^{14}}{(1.4176 \cdot 10^{8} \angle 6.4^\circ) \cdot (6.4126 \cdot 10^{8} \angle 13.5^\circ)}$$

$$H_{C,4}(j1{,}600\pi) = 5.9087 \cdot 10^{-3} \angle -19.9^\circ$$

This indicates a gain of –44.57 dB at the stopband frequency, which exceeds the specification, and a phase shift of –19.9° or 340.1°. In the case of the passband frequency of 2,000 Hz, similar calculations can be made as shown below:

$$H_{C,4}(j4{,}000\pi) = \frac{2.0982 \cdot 10^{16}}{(8.2263 \cdot 10^{6} + j3.9585 \cdot 10^{7}) \cdot (4.9099 \cdot 10^{8} + j3.7326 \cdot 10^{8})}$$

$$H_{C,4}(j4{,}000\pi) = \frac{2.0982 \cdot 10^{16}}{(4.0431 \cdot 10^{7} \angle 78.3^\circ) \cdot (6.1676 \cdot 10^{8} \angle 37.2^\circ)}$$

$$H_{C,4}(j4{,}000\pi) = 0.8414 \angle -115.5^\circ$$

This indicates a gain of –1.50 dB at the stopband frequency, which meets the specification, and a phase shift of –115.5° or 244.50°.

3.5.2 C Code for Frequency Response Calculation

We are now ready to develop the C code for determining the frequency response of an analog filter. In order to properly determine the frequency response, we need to know the starting and stopping frequencies for our calculations. We also need to know whether to space the frequencies in a linear or logarithmic fashion, and whether to calculate the magnitude in decibels or not. This, as well as other, information is stored in a frequency response structure called `Resp_Params`. All of the functions that actually perform the frequency response calculations are contained in the F_RESPON.C module that has a header file of F_RESPON.H in which `Resp_Params` is defined.

The primary function for the calculation of the analog frequency response is `Calc_Analog_Resp,` which is shown in Listing 3.1. (The set of frequencies used to evaluate the filter have already been calculated and stored in `RP->freq`.)

```
/*=====================================================
  Calc_Analog_Resp() - calcs response for analog filts
  Prototype:  int Calc_Analog_Resp(Filt_Params *FP,
                                  Resp_Params *RP);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
              RP - ptr to struct holding respon params
=====================================================*/
int Calc_Analog_Resp(Filt_Params *FP,Resp_Params *RP)
{ int     c,f,q;        /*  loop counters */
  double  rad2deg,      /*  rad to deg conversion */
          omega,omega2, /*  radian freq and square */
          rea,img;      /*  real and imag part */

  rad2deg = 180.0 / PI; /*  set rad2deg */
  /*  Loop through each of the frequencies */
  for(f = 0 ;f < RP->tot_pts; f++)
  { /*  Initialize magna and angle */
    RP->magna[f] = FP->gain;
    RP->angle[f] = 0.0;
    /*  Pre calc omega and omega squared */
    omega = PI2 * RP->freq[f];
    omega2 = omega * omega;
    /*  Loop through coefs for each quadratic */
    for(q = 0 ;q < (FP->order+1)/2; q++)
    { /*  c is coef index = 3 * quad index */
      c = q * 3;
      /* Numerator values */
      rea = FP->acoefs[c+2] - FP->acoefs[c] * omega2;
      img = FP->acoefs[c+1] * omega;
      RP->magna[f] *= sqrt(rea*rea + img*img);
      RP->angle[f] += atan2(img,rea);
      /* Denominator values */
      rea = FP->bcoefs[c+2] - FP->bcoefs[c] * omega2;
      img = FP->bcoefs[c+1] * omega;
      RP->magna[f] /= sqrt(rea*rea + img*img);
      RP->angle[f] -= atan2(img,rea);
    }
    /* Convert to degrees */
    RP->angle[f] *= rad2deg;
  }
  /*  Convert magnitude response to dB if indicated */
  if(RP->mag_axis == LOG)
  { for(f = 0 ;f < RP->tot_pts; f++)
    { /* Handle very small numbers */
      if(RP->magna[f] < ZERO)
      { RP->magna[f] = ZERO;}
      RP->magna[f] = 20 * log10(RP->magna[f]);
    }
  }
  return ERR_NONE;
}
```

Listing 3.1 `Calc_Analog_Resp` **function.**

After calculating the constant `rad2deg` for converting radians to degrees, the primary work of the function is performed within two nested `for` loops. The outer loop controls the frequency at which the response is being calculated, while the inner loop steps through the number of quadratics in the approximation function. As we start the calculation for a new frequency, the magnitude value (`RP->magna[f]`) is initialized to the overall gain value of the filter, while the phase value (`RP->angle[f]`) is set to 0. We then convert to radian frequencies and make the calculation of the radian frequency squared outside the quadratic loop. That saves time by not repeatedly making those calculations inside the loop where there is no change in their value.

Once we reach the inner quadratic loop, which is controlled by the variable `q`, we define a value `c` that is based on `q` to help access the individual coefficients of each quadratic. We first handle the numerator portion of the quadratic by determining the real and imaginary parts of the complex number. Then the total magnitude is multiplied by the magnitude of the complex number, and the total angle is incremented by the angle of the complex number.

A similar process is performed for the denominator portion of the quadratic except that the total magnitude is divided by the denominator's magnitude, and the denominator's phase is subtracted from the total phase. This inner loop process is repeated for all of the quadratic terms, and then the phase value is converted to degrees outside the quadratic loop.

If the magnitude of the response was specified to be in decibels, each of the magnitude values is converted to decibels before leaving the function. Since the logarithm of zero is undefined, an artificial value `ZERO` is defined in F_RESPON.H so that no math error will be produced by the compiler if values become too small. Since we are using type `double` to describe the magnitude values, a value of 1E-30 was chosen for `ZERO` that is still within the limits of expression for doubles. This value will produce a gain value of –600 dB, which is well beyond the normal levels of gain we expect in filter design, so our definition of `ZERO` should have no effect on the normal operation of the program.

3.6 SAVING THE FILTER PARAMETERS

Once a filter has been designed, WFilter can save it. The saved file can then be used by other software programs to identify the values of the filter gain, coefficients, and other characteristics of the filter, or the filter can be reloaded into WFilter for continued work. The structure of this WFilter binary data file is therefore included below.

A binary file has a number of advantages over a text file in this situation. First, the binary file will be easier to read if the data is needed by another program. There is a very good chance that the filter designer will want to transfer the filter coefficients that have been determined by WFilter to another program for testing or implementation. Second, the coefficients and data can be stored with more accuracy than they can be displayed.

In order to better understand the binary file structure, the format of the file is shown below. The first 4 bytes in the file will be an acronym for **A**nalog and **D**igital **F**ilter **D**esign. The next byte is a *w* to indicate that the file was created by WFilter. The next 3 bytes are characters indicating the implementation, approximation, and selectivity types. The next 40 bytes are dedicated to the text description for the filter. The next 8 bytes are separated into 6 bytes of reserved space to allow for future indicators, and 2 bytes used to store the order of the filter. The filter order is placed into the binary buffer in such a manner that it could be read as an integer if desired. (The PC format for disk storage dictates that the low-order byte is written before the high-order byte for an integer. Other machines may use other methods, but will be consistent between read and write operations.) Next, 10 variables of size `double` are stored, which includes the sampling frequency, the gains and frequencies in the order indicated in the listing, and the gain of the filter. Finally, the `acoefs` and `bcoefs` variables are written as type `double` in the order that they were stored by the program.

Header (8 bytes) — Contains the identification “ADFDw” followed by three characters indicating the implementation, approximation, and selectivity types, respectively.

The implementation options and values are:
0 - analog, 1 - digital FIR and 2 - digital IIR.

The approximation options and values for analog and digital IIR filters are:
0 - Butterworth, 1 - Chebyshev, 2 - inverse Chebyshev, and
3 - elliptic.

The approximation (or window) options and values for digital FIR filters are:
4 - rectangular, 5 - Barlett, 6 - Blackman, 7 - Hamming, 8 - von Hann,
9 - Kaiser, and 10 - Parks-McClellan. (Discussion of the digital approximations will be given later in the text.)

The selectivity options and values are: 0 - lowpass, 1 - highpass, 2 - bandpass, and 3 - bandstop.

Description (40 bytes) — Contains the text description of the filter and is filled by nulls at the end of the description to make exactly 40 characters.

Reserved (6 bytes) — Reserved for future use.

Order (2 bytes) — The order can be read as an integer.

Specifications (10 doubles) — Contains the sampling frequency in hertz, followed by the gains and frequencies from the specifications, and finally the overall filter gain. The gains are specified in decibels and are in the order of a_{pass1}, a_{pass2}, a_{stop1}, and a_{stop2}. The frequencies are specified in hertz and are in the order of f_{pass1}, f_{pass2}, f_{stop1}, and f_{stop2}.

A coefficients (variable) — The acoefs written as doubles.

B coefficients (variable) — The bcoefs written as doubles.

3.7 CONCLUSION

We have now completed the design of a number of analog approximation types and are able to calculate the frequency response and save the filter parameters for future use. In the next chapter we'll see how to implement these functions as active filters. For those interested in the C code used to unnormalize the filter transfer functions, please turn to Appendix E.

Chapter 4

Analog Filter Implementation Using Active Filters

In this chapter, we will discuss the implementation of the analog approximation functions that we developed and verified in previous chapters. There are several methods that could be used to allow the successful implementation of an analog filter. Unfortunately, an entire book could be devoted to this topic (and several have), so we will concentrate on active filters. Using active filters to implement the transfer functions is a very popular method today because of the natural correspondence between the analog circuit and the mathematical function. We will not discuss the derivation of the transfer functions for active filters because the development of such circuit analysis techniques is beyond the scope of this text. However, a number of suitable references are given in the analog active filter texts of Appendix A for those who are interested in the derivations. Instead, the circuit topology and transfer function for several common active filters will be presented before determining the component values for each circuit. We will develop implementation procedures for each of the filter selectivities discussed in previous chapters as well as discuss other options for implementation and some of the important implementation issues.

4.1 IMPLEMENTATION PROCEDURES FOR ANALOG FILTERS

There are many choices when it comes to implementing our continuous-time analog filters. For example, we could describe our single-input and single-output system by means of state variables and use a state-space approach to the problem. This method has the advantage of being very general and it puts the problem in a convenient mathematical form that allows manipulation in terms of matrix algebra. However, the development of this theory is beyond the scope of this text, but several references in Appendix A provide insight to this method.

When considering the method of implementation, two key factors of frequency range and the power handling capability of the filter also come into question. Generally speaking, high power and high frequency dictates that passive filters be used. Active filters (with op-amps) have a maximum frequency range and power range where they can be used successfully. (Section 4.7 discusses the frequency issue in more detail.) Passive filters are also a bit more of a challenge to design because they do not exhibit the near ideal qualities of input and output impedances that op-amps have.

Another choice for filter implementation is the transconductance-C (Gm-C) filter that extends the effective frequency range beyond that of the typical active filter. Finally, switched-capacitor (SC) filters, like the Gm-C filters, use MOSFET technology and switching signals to implement analog filters. In that way the SC filter uses technology that bridges the gap between continuous-time and discrete-time systems.

However, it is not possible to treat the complete field of analog filter design in one chapter. Instead, we will concentrate on a more traditional approach to implementing our transfer functions using a single form of active filter. The Sallen-Key filter, which will be discussed in more detail in the following sections, has the advantage of implementing a second-order factor with a single op-amp stage. There are many other topologies (circuit configurations) that could be chosen, but the Sallen-Key is a tried and true configuration. It is important to see one method of implementing our continuous-time transfer function before beginning the discussion of discrete-time systems.

Each of the active filters discussed will be composed of several stages of electronic circuitry consisting of a single operational amplifier (op-amp) and a number of electronic components called resistors (*R*s) and capacitors (*C*s). The fact that active filters can implement complex poles without the use of inductors (*Ls)* is a key point in their favor. Inductors have the disadvantages of being large, heavy, costly, and generators of spurious magnetic fields. Therefore, being able to implement an active filter with components that can be miniaturized is a big advantage.

Each of these stages of electronic filtering will have a transfer function that characterizes the relationship of the output voltage to the input voltage, as indicated in (4.1). More specifically, each quadratic factor of the transfer function will have the form as shown in (4.2), where each *A* and *B* in the transfer function will be a function of the *R* and *C* used in the circuit. (The subscript *c* is used to indicate that these transfer functions are describing the *circuit* response.)

$$H_c(s) = V_o(s)/V_i(s) \tag{4.1}$$

$$H_c(s) = \frac{A_o \cdot s^2 + A_1 \cdot s + A_2}{B_o \cdot s^2 + B_1 \cdot s + B_2} \tag{4.2}$$

We also have the approximation function that we derived to meet a specific set of frequency and attenuation characteristics. The general form of each quadratic factor in the approximation function is shown in (4.3), where the subscript *a* is used to designate this quadratic factor as an *approximation* function. The *a*s and *b*s used as coefficients have already been determined and are numerical constants.

$$H_a(s) = \frac{a_o \cdot s^2 + a_1 \cdot s + a_2}{b_o \cdot s^2 + b_1 \cdot s + b_2} \tag{4.3}$$

The implementation process then becomes a matter of equating the two transfer function factors as shown below. Each *A* and *B* of the circuit transfer function are equated to the respective *a* and *b* of the approximation function and results in several equations which must be solved for appropriate *R* and *C* values.

$$H_c(s) = H_a(s) \tag{4.4}$$

We will see this common procedure used throughout the next four sections as we determine the component values needed to implement each of the active filters.

4.2 LOWPASS ACTIVE FILTERS USING OP-AMPS

There are a number of active filter topologies (circuit configurations) that could be used to implement a lowpass filter. We will limit ourselves to the popular Sallen-Key filter (shown in Figure 4.1) with a transfer function as described in (4.5). As indicated by the transfer function, this active filter stage can implement one second-order factor of a lowpass filter function. This form is very convenient since it naturally implements the quadratic factor that we have been using for the description of the approximation function. Of course, several of these circuit stages in cascade can be used to implement higher-order functions, and with the addition of a single first-order stage, odd-order filters can be implemented as well.

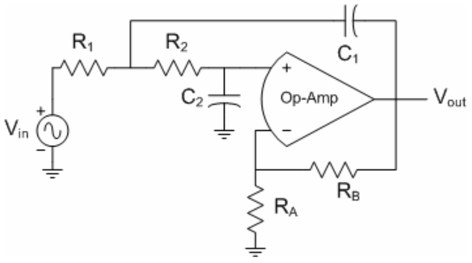


Figure 4.1 Sallen-Key lowpass active filter stage.

$$H_{c,L}(s) = \frac{K / R_1 R_2 C_1 C_2}{s^2 + \left[1/R_1C_1 + 1/R_2C_1 + (1-K)/R_2C_2\right] \cdot s + 1/R_1R_2C_1C_2} \quad (4.5)$$

where

$$K = 1 + (R_B / R_A) \quad (4.6)$$

In comparison to the transfer function of (4.5), (4.7) shows the general form of an all-pole lowpass approximation function such as the Butterworth and Chebyshev functions. (Inverse Chebyshev and elliptic filter approximations will use a different active filter to implement them since they require zeros in the complex plane.) The numerator value a_2 in (4.7) will always be equal to b_2, and G will have a value of unity except for the even-order Chebyshev case.

$$H_{a,L}(s) = \frac{G \cdot a_2}{s^2 + b_1 \cdot s + b_2} \quad (4.7)$$

If we equate (4.5) and (4.7) as indicated in the previous section, we would generate one equation for b_1 and one equation for b_2 (or a_2). However, we have five unknowns (R_1, R_2, C_1, C_2, and K) in our circuit. Since there are more unknowns than equations, we cannot uniquely determine the values of the components required for the active filter. This allows us to select values for three of the components. For example, we could select the values of R_1, R_2, and C_2 if we wanted or the values of C_1, C_2, and K. Another method that we will use is to let $R_1 = R_2$ and $C_1 = C_2$, which effectively eliminates two of the unknowns. In addition, we will pick a common value for the capacitors since they have far fewer available values than resistors. This method of picking the R and C values has the advantage of reducing the number of different component values needed to implement the active filter. The disadvantage of this method is that the value of K, which represents the gain for the active filter, will not be unity. However, we will find that this problem can be easily solved.

If we select common values for the primary resistors and capacitors, the transfer function for the active filter then becomes

$$H_{c,L}(s) = K \cdot \frac{1/R^2C^2}{s^2 + \left[(3-K)/RC\right] \cdot s + 1/R^2C^2} \quad (4.8)$$

By considering the denominators of (4.7) and (4.8), we can determine the two relationships as shown below:

$$b_2 = 1/R^2C^2 \quad (4.9)$$

$$b_1 = (3 - K)/RC \tag{4.10}$$

Since we have already picked the value of *C*, we can solve for the resistor values needed to implement the filter. The results are

$$R = 1/\sqrt{b_2 C^2} \tag{4.11}$$

$$K = 3 - \left(b_1 / \sqrt{b_2}\right) \tag{4.12}$$

and then using (4.6)

$$R_B / R_A = 2 - \left(b_1 / \sqrt{b_2}\right) \tag{4.13}$$

Usually, R_A will be picked as some convenient value and R_B will then be calculated.

The only adjustment remaining is to equate the numerator terms of the two transfer functions. Notice that in this case, there exists a gain of *K* at ω = 0, while the approximation function has a gain of *G*. The value of *G* will always be less than or equal to one, while the value *K* will always be greater than or equal to one. Thus, it is always necessary to reduce the gain of the active filter to match the gain of the approximation function. The amount of this gain adjustment factor is

$$GA = K_{\text{tot}} / G_{\text{tot}} \tag{4.14}$$

where K_{tot} and G_{tot} represent the products of all the *K*s and *G*s in the total circuit and approximation functions, respectively. This gain adjustment factor can be implemented by a simple voltage divider at the output of the active filter stage. There are two conditions placed on this voltage divider. The inverse of the voltage divider ratio must match the gain adjustment factor, and the equivalent resistance of the voltage divider as seen by the output load must equal the required filter output resistance. Therefore, assuming an output resistance R_{out}, and a voltage divider network made up of R_x and R_y,

$$R_{\text{out}} = R_x \cdot R_y /(R_x + R_y) \tag{4.15}$$

$$GA = (R_x + R_y) / R_y \tag{4.16}$$

which means that

$$R_x = GA \cdot R_{\text{out}} \tag{4.17}$$

$$R_y = GA \cdot R_{\text{out}} /(GA - 1) \tag{4.18}$$

The circuit of Figure 4.2 shows one stage of the new configuration with the voltage divider output. If the input signal level is very high, we might choose to use the voltage divider on the first stage of the filter in order to reduce distortion in the filter.

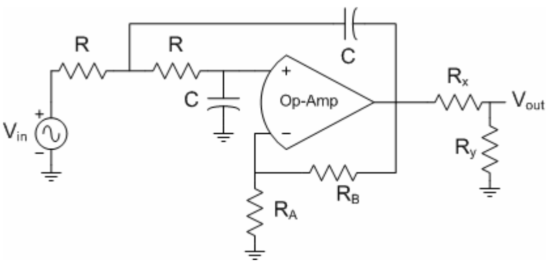


Figure 4.2 Active filter with voltage divider output.

Before we consider how to implement an odd-order approximation function, we need to consider whether the gain adjustment technique is appropriate for all applications. In many cases, amplification is a required part of the filtering system, and therefore the differences in gain can be factored into the overall gain requirement. For example, if the overall system gain required is 60 dB, and the filter is producing a gain of 20 dB in the passband, a more efficient solution would be to design the remaining amplifiers to provide 40 dB of gain. This design would be more power efficient, use fewer components, and have better dynamic range. However, the preceding technique can be used if an exact gain is required from the filter.

If an odd-order approximation factor is to be implemented, an odd-order stage as shown in Figure 4.3 can be used as the first stage of the active filter. This simple RC filter is followed by a buffer amp so that the output impedance of the RC combination will not affect the input to the next active filter stage. However, if an op-amp is going to be required to implement this first-order factor, we might consider adding a few more components and implementing a second-order stage instead. In many cases, additional attenuation in the stopband would be welcome, and the additional cost is slight.

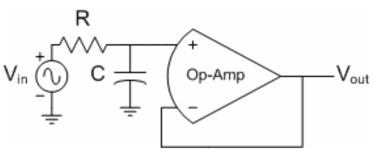


Figure 4.3 First-order lowpass filter with buffer amp.

The transfer function for the first-order stage is given in (4.19). This transfer function must match a first-order approximation function as given in (4.20):

$$H_c(s) = \frac{1/RC}{s + 1/RC} \tag{4.19}$$

$$H_a(s) = \frac{G \cdot a_2}{s + b_2} \tag{4.20}$$

Again, the values of a_2 and b_2 will be identical and G will have a value of 1. Then using the value of C that has already been picked,

$$R = 1/b_2C \tag{4.21}$$

Example 4.1 illustrates how to use the information presented in this section to design an active lowpass filter.

Example 4.1 Butterworth Lowpass Active Filter Design

Problem: Determine the resistor and capacitor values to implement a Butterworth lowpass active filter to meet the following specifications:

$a_{\text{pass}} = -1$ dB, $a_{\text{stop}} = -50$ dB, $f_{\text{pass}} = 1000$ Hz, and $f_{\text{stop}} = 4000$ Hz

Solution: First, we find that a fifth-order filter is required. The resulting unnormalized approximation function can then be determined to be

$$H_a(s) = \frac{7{,}192.2 \cdot (51.728 \cdot 10^6)^2}{(s + 7{,}192.2) \cdot (s^2 + 4{,}445.0 \cdot s + 51.728 \cdot 10^6) \cdot (s^2 + 11{,}637 \cdot s + 51.728 \cdot 10^6)}$$

The three factors in the denominator can now be matched to three active filter stages. By picking $C = 0.01$ μF and $R_A = 10$ kΩ, the remaining values for each stage can be calculated from (4.11)–(4.13) and (4.21). The first-order stage does not require an R_B value.

m	R_m	K_m	R_{Bm}
0	13.90 kΩ	—	—
1	13.90 kΩ	2.3820	13.82 kΩ
2	13.90 kΩ	1.3820	3.820 kΩ

In order to achieve a gain of exactly 1 at ω = 0, we can use a voltage divider at the output. First, we determine K_{tot}

$$K_{\text{tot}} = \prod_m K_m = 3.2919$$

This value along with the fact that G_{tot} has a value of 1 allows us to determine that $GA = 3.2919$. The following resistor divider values can then be determined assuming a required output impedance of 10 kΩ. (The gain adjustment could be factored into the overall gain of the system instead of using the voltage divider.)

$$R_x = 32.92\ \text{k}\Omega, \quad R_y = 14.36\ \text{k}\Omega$$

The resulting active filter is shown in Figure 4.4. We will generate the C code for the determination of these component values in a later section of this chapter.

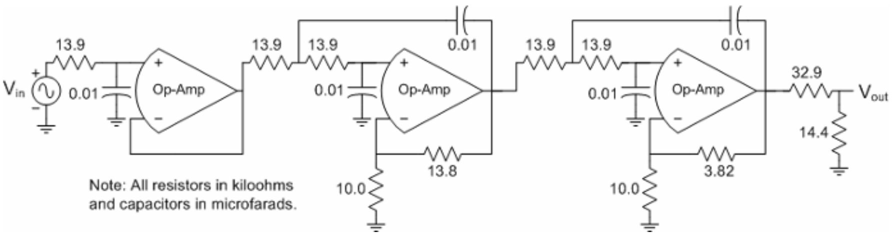


Figure 4.4 Butterworth lowpass active filter for Example 4.1.

4.3 HIGHPASS ACTIVE FILTERS USING OP-AMPS

We will also use a Sallen-Key circuit configuration for implementing a highpass filter. The circuit for implementing a second-order factor is shown in Figure 4.5, with the transfer function for the stage shown in (4.22). We can see that this highpass configuration is the same as the lowpass case except that the primary *R*s and *C*s have changed positions, although R_A and R_B remain in their original positions.

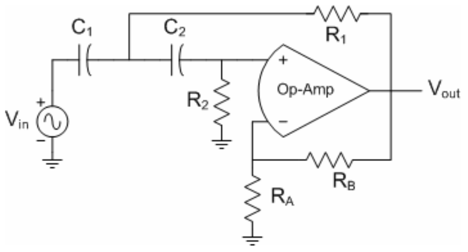


Figure 4.5 Sallen-Key highpass active filter stage.

$$H_{c,H}(s) = \frac{K \cdot s^2}{s^2 + [1/R_2C_2 + 1/R_2C_1 + (1-K)/R_1C_1] \cdot s + 1/R_1R_2C_1C_2} \tag{4.22}$$

where again

$$K = 1 + (R_B / R_A) \tag{4.23}$$

We can use the same procedure in selecting the component values as we used in the lowpass case by letting $R_1 = R_2$ and $C_1 = C_2$. The result is shown in (4.24), where we see that the denominator is identical to the lowpass case in (4.8):

$$H_{c,H}(s) = \frac{K \cdot s^2}{s^2 + [(3-K)/RC] \cdot s + 1/R^2C^2} \tag{4.24}$$

This equation can then be matched to the quadratic factors from the approximation functions that have the form shown in (4.25)

$$H_{a,H}(s) = \frac{G \cdot s^2}{s^2 + b_1 \cdot s + b_2} \tag{4.25}$$

When the equation for the approximation function is compared to the equation for the active filter stage, we see that the relationships between terms are identical to those of the lowpass case. Therefore, if we pick the same value of capacitor, the resistor values will be identical to those of the lowpass case, as shown below:

$$R = 1/\sqrt{b_2C^2} \tag{4.26}$$

$$R_B / R_A = 2 - \left(b_1 / \sqrt{b_2}\right) \tag{4.27}$$

The adjustment of gain for the highpass case is handled in a similar manner to the lowpass case. As (4.24) indicates, the K value of each stage can be determined by allowing the frequency to approach infinity, as opposed to zero in the lowpass case. K_{tot} can then be easily determined, and with the approximation function gain, the gain adjustment factor GA can be determined as shown previously in (4.14). A resistive voltage divider is used at the output of the last stage of the active filter. If necessary, a buffer amplifier can be used after this voltage divider if the impedance of the network being driven by the filter is too small.

If an odd-order highpass approximation factor is to be implemented, an active filter stage as shown in Figure 4.6 can be used as the first stage of the active filter. The only difference between this stage and the lowpass first-order stage is the interchange of R and C values.

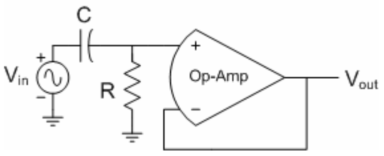


Figure 4.6 First-order highpass filter with buffer amp.

The transfer function for this stage is given in (4.28), while the approximation function for a first-order highpass factor is shown in (4.29):

$$H_c(s) = \frac{s}{s + 1/RC} \tag{4.28}$$

$$H_a(s) = \frac{G \cdot s}{s + b_2} \tag{4.29}$$

As in the second-order case, the resistor value will be the same as the lowpass value assuming that the same value of capacitor is picked.

$$R = 1/b_2 C \tag{4.30}$$

Example 4.2 Chebyshev Highpass Active Filter Design

Problem: Determine the resistor and capacitor values to implement a Chebyshev highpass active filter to meet the following specifications:

$a_{\text{pass}} = -0.5$ dB, $a_{\text{stop}} = -30$ dB, $f_{\text{pass}} = 1000$ Hz, and $f_{\text{stop}} = 400$ Hz

Solution: A fourth-order transfer function as shown below is required. Since this is an even-order Chebyshev, there is an adjustment factor of 0.94406 included in the numerator.

$$H_a(s) = \frac{.94406 \cdot s^4}{(s^2 + 2{,}071.9 \cdot s + 37.121 \cdot 10^6) \cdot (s^2 + 14{,}926 \cdot s + 110.77 \cdot 10^6)}$$

The two quadratic terms can be matched to two active filter stages by again picking $C = 0.01$ μF and $R_A = 10$ kΩ. The other circuit values can be calculated as shown below:

m	R_m	K_m	R_{Bm}
0	16.41 kΩ	2.6599	16.6 kΩ
1	9.502 kΩ	1.5818	5.82 kΩ

In order to achieve a gain of exactly 1 at ω = ∞, we can use a voltage divider at the output. First, we determine the gain adjustment factor for the filter that in this case includes not only the K_m factors for each quadratic, but also the approximation function's gain of 0.94406:

$$GA = (\prod_m K_m) / 0.94406 = 4.4567$$

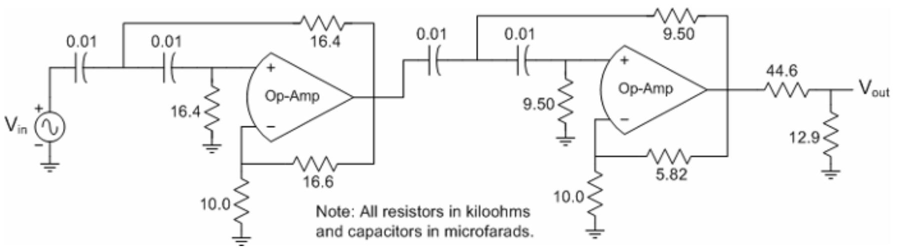


Figure 4.7 Chebyshev highpass active filter for Example 4.2.

Assuming an equivalent resistance of 10 kΩ, the resistor divider values can be determined with the resulting filter shown in Figure 4.7. (The gain adjustment could be factored into the overall gain of the system instead of using the voltage divider.)

$$R_x = 44.57 \text{ k}\Omega, \quad R_y = 12.89 \text{ k}\Omega$$

4.4 BANDPASS ACTIVE FILTERS USING OP-AMPS

Figure 4.8 shows a Sallen-Key active filter stage that implements a second-order bandpass transfer function. The transfer function for this bandpass stage is given in (4.31).

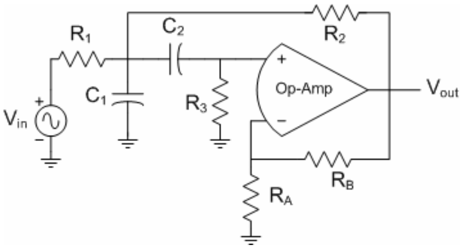


Figure 4.8 Sallen-Key bandpass active filter stage.

$$H_{c,P}(s)=\frac{K\cdot s/R_1C_1}{s^2+\left[\frac{1}{R_1C_1}+\frac{1}{R_3C_1}+\frac{1}{R_3C_2}+\frac{(1-K)}{R_2C_1}\right]\cdot s+\frac{R_1+R_2}{R_1R_2R_3C_1C_2}} \tag{4.31}$$

where again

$$K=1+(R_B/R_A) \tag{4.32}$$

We can simplify this function by letting R_1 = R_2 = R_3 and C_1 = C_2. This simplified function is given in (4.33):

$$H_{c,L}(s)=\frac{K\cdot s/RC}{s^2+[(4-K)/RC]\cdot s+2/R^2C^2} \tag{4.33}$$

This bandpass transfer function can now be matched to the general form of the approximation factor, as shown in (4.34):

$$H_{a,H}(s)=\frac{a_1\cdot s}{s^2+b_1\cdot s+b_2} \tag{4.34}$$

After matching equivalent denominator terms, the following equations emerge,

$$b_1 = (4 - K)/RC \tag{4.35}$$

$$b_2 = 2/R^2C^2 \tag{4.36}$$

which leads to

$$R = \sqrt{2/b_2C^2} \tag{4.37}$$

$$K = 4 - \sqrt{2b_1^2/b_2} \tag{4.38}$$

$$R_B/R_A = 3 - \sqrt{2b_1^2/b_2} \tag{4.39}$$

After these calculations are made, the overall gain of the active filter must be determined. This is a more difficult task than for the lowpass and highpass cases since the gain adjustment must be determined at the center frequency of the passband ω_o. If we refer again to (4.33) and (4.34), we see that they will have identical denominator coefficients since we have just derived the equations to guarantee that. However, the numerators differ by the constants that are present. The gain adjustment factor for each stage is then the ratio of the two numerator constants, as shown in (4.40). The total gain adjustment is then the product of these stage gain adjustments:

$$GA = \prod_m K_m /(a_{1m}R_mC_m) \tag{4.40}$$

Once the total gain adjustment has been determined, a voltage divider stage at the output of the active filter can be used for compensation.

Example 4.3 Butterworth Bandpass Active Filter Design

Problem: Determine the resistor and capacitor values to implement a Butterworth bandpass active filter to meet the following specifications:

a_{pass} = –1.5 dB,	a_{stop} = –28 dB,	f_{pass1} = 1,000 Hz,
f_{pass2} = 2,000 Hz,	f_{stop1} = 500 Hz, and	f_{stop2} = 4,000 Hz

Solution: A third-order equivalent lowpass filter is needed, which indicates that a sixth-order bandpass function will result as shown below:

$$H_a(s) = \frac{(7282 \cdot s)^3}{(s^2 + 7{,}282 \cdot s + 78.96 \cdot 10^6) \cdot (s^2 + 2{,}402 \cdot s + 38.88 \cdot 10^6) \cdot (s^2 + 4{,}880 \cdot s + 160.3 \cdot 10^6)}$$

By picking C = 0.01 μF and R_A = 10 kΩ, the remaining values can be determined by matching the three quadratic terms:

m	R_m	K_m	R_{Bm}	GA_m
0	15.92 kΩ	2.8410	18.41 kΩ	2.4512
1	22.68 kΩ	3.4550	24.55 kΩ	2.0919
2	11.17 kΩ	3.4550	24.55 kΩ	4.2481

The combined gain produced by the active filter stages at $\omega = \omega_o$ is determined to be 21.783, which can be compensated by a voltage divider at the output of the filter. If the output resistance is to be 10 kΩ, the voltage divider resistor values are given below. (Note that in this case, if a voltage divider were used, a gain of over 21 (over 40 dB) would be sacrificed. It would be better to include this in the system gain.)

$$R_x = 218\,\text{k}\Omega, \quad R_y = 10.5\,\text{k}\Omega$$

The resulting bandpass filter is shown in Figure 4.9.

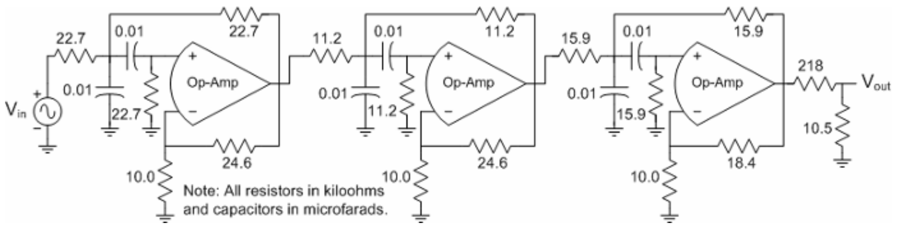


Figure 4.9 Butterworth bandpass active filter for Example 4.3.

4.5 BANDSTOP ACTIVE FILTERS USING OP-AMPS

Figure 4.10 shows a twin-tee bandstop active filter stage that can implement a variety of second-order functions. The admittance labeled Y can represent a conductance G, or a susceptance sC, or can have zero value (not be present at all).

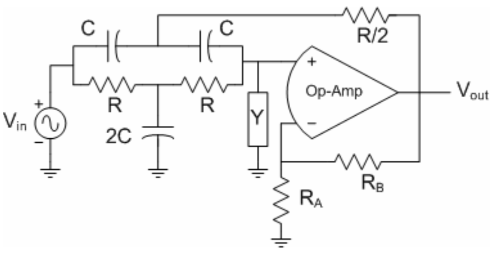


Figure 4.10 Twin-tee bandstop active filter stage.

The general form of the transfer function for this filter is given in (4.41):

$$H_{c,S}(s) = \frac{K \cdot \left[s^2 + \dfrac{1}{R^2C^2} \right]}{s^2 + \left[\dfrac{4 - 2K + 2RY}{RC} \right] \cdot s + \dfrac{1 + 2RY}{R^2C^2}} \tag{4.41}$$

where again

$$K = 1 + (R_B / R_A) \tag{4.42}$$

Equation (4.43) rewrites this equation in terms of the pole frequency ω_p and the zero frequency ω_Z:

$$H_{c,S}(s) = \frac{K \cdot (s^2 + \omega_z^2)}{s^2 + (\omega_p / Q) \cdot s + \omega_p^2} \tag{4.43}$$

The transfer function of (4.43) must be matched to the general form of a bandstop approximation function, as shown in (4.44):

$$H_{a,S}(s) = \frac{G \cdot (s^2 + a_2)}{s^2 + b_1 \cdot s + b_2} \tag{4.44}$$

Depending on the value of *Y* selected, ω_Z may be greater than, equal to, or less than ω_p. This will affect the matching of the respective terms in (4.43) and (4.44). The responses for the transfer function will also change, as indicated in Figure 4.11. Let's look at how the transfer function changes for the three cases.

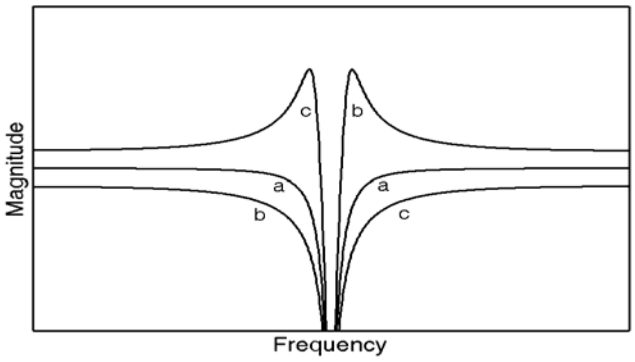


Figure 4.11 Bandstop filter responses.

First, we consider the case where no *Y* element is present (*Y* = 0), in which case the transfer function can be simplified as shown in (4.45). Note that this function has the condition that $\omega_z = \omega_p$ ($a_2 = b_2$), which will produce a bandstop response as shown in Figure 4.11(a). Notice that the response has the condition that the upper and lower passbands have equal gain.

$$H_{c,S0}(s) = \frac{K \cdot \left[s^2 + \dfrac{1}{R^2C^2} \right]}{s^2 + \left[\dfrac{4-2K}{RC} \right] \cdot s + \dfrac{1}{R^2C^2}} \qquad (4.45)$$

The selection of components to implement this type of response is relatively easy as we match the terms of (4.44) and (4.45). By first picking a value of *C*, the results are

$$R = 1 \Big/ \sqrt{a_2 C^2} \qquad (4.46)$$

$$K = 2 - \sqrt{b_1^2 / 4b_2} \qquad (4.47)$$

$$R_B / R_A = 1 - \sqrt{b_1^2 / 4b_2} \qquad (4.48)$$

Next, in the case where $Y = G_0 = 1/R_0$, the resulting transfer function is given in (4.49). Note that this function has $\omega_z < \omega_p$ ($a_2 < b_2$), which will produce a bandstop response as shown in Figure 4.11(b). This is sometimes referred to as a "highpass notch" filter because the gain of the upper passband is larger than that of the lower passband.

$$H_{c,S1}(s) = \frac{K \cdot \left[s^2 + \dfrac{1}{R^2C^2} \right]}{s^2 + \left[\dfrac{(4 - 2K + 2R/R_o)}{RC} \right] \cdot s + \dfrac{(R_o + 2R)/R_o}{R^2C^2}} \tag{4.49}$$

The selection of components to implement this type of response is again determined by matching terms and by first picking a value of C; the results are

$$R = 1\Big/\sqrt{a_2C^2} \tag{4.50}$$

$$R_o = 2R\Big/\left[\left(b_2/a_2\right) - 1\right] \tag{4.51}$$

$$K = 2 + (b_2 - a_2 - b_1\sqrt{a_2})\Big/(2a_2) \tag{4.52}$$

$$R_B/R_A = 1 + (b_2 - a_2 - b_1\sqrt{a_2})\Big/(2a_2) \tag{4.53}$$

Finally, in the case where $Y = sC_o$, the resulting transfer function is given in (4.54). Note that this function has $\omega_z > \omega_p$ ($a_2 > b_2$), which will produce a bandstop response as shown in Figure 4.11(c). This is sometimes referred to as a "lowpass notch" filter.

$$H_{c,S2}(s) = \frac{\dfrac{K \cdot C}{(C + 2C_o)} \cdot \left[s^2 + \dfrac{1}{R^2C^2} \right]}{s^2 + \left[\dfrac{(4 - 2K) \cdot C + 2C_o}{RC \cdot (C + 2C_o)} \right] \cdot s + \dfrac{C}{R^2C^2 \cdot (C + 2C_o)}} \tag{4.54}$$

The selection of components to implement this type of response is again determined by matching terms and by first picking a value of C; the results are

$$R = 1 \Big/ \sqrt{a_2 C^2} \tag{4.55}$$

$$C_o = C \cdot \left[(a_2 / b_2) - 1 \right] / 2 \tag{4.56}$$

$$K = 2 + (a_2 - b_2 - b_1 \sqrt{a_2}) \Big/ (2b_2) \tag{4.57}$$

$$R_B / R_A = 1 + (a_2 - b_2 - b_1 \sqrt{a_2}) \Big/ (2b_2) \tag{4.58}$$

After these calculations are made, the overall gain of the active filter can be determined by evaluating the transfer function at $\omega = 0$ or $\omega = \infty$. If the gain of the circuit is to be determined at $\omega = 0$, then the value of K for that stage is used except for the highpass notch stage. In that case, an additional multiplication factor of R_o / (R_o + 2R) should be included as seen from (4.49). If the gain is to be determined at an infinite frequency, then the lowpass notch stage will have a gain that must be increased by a value of C / (C + 2C_o), as indicated by (4.54). These values will be the same and can be included in the circuit using a voltage divider.

Example 4.4 Chebyshev Bandstop Active Filter Design

Problem: Determine the resistor and capacitor values to implement a Chebyshev bandstop active filter to meet the following specifications:

a_{pass} = –1 dB,	a_{stop} = –40 dB,	f_{pass1} = 666.67 Hz,
f_{pass2} = 1,500 Hz,	f_{stop1} = 909.09 Hz, and	f_{stop2} = 1,100 Hz

Solution: A third-order lowpass equivalent function is required, which indicates that a sixth-order unnormalized bandstop approximation function will be necessary, as shown below:

$$H_a(s) = \frac{(s^2 + 39.48 \cdot 10^6)^3}{(s^2 + 10{,}600 \cdot s + 39.48 \cdot 10^6)(s^2 + 811.0 \cdot s + 17.87 \cdot 10^6)(s^2 + 1{,}792 \cdot s + 87.21 \cdot 10^6)}$$

By picking C = 0.01 μF and R_A = 10 kΩ, the remaining values can be calculated by matching terms. Note that there are three denominator quadratics with three different values of constant terms. One of these values is larger than the numerator constant term, one is equal to the numerator constant term, and one is smaller than the numerator constant term. This indicates that one of the stages will have an R_o value, one will have no Y element, and one will have a C_o value, respectively.

m	R_m	K_m	R_{Bm}	Y_m
0	15.92 kΩ	1.1569	1.569 kΩ	—
1	15.92 kΩ	2.4619	14.62 kΩ	C_o = 6.045 nF
2	15.92 kΩ	2.4619	14.62 kΩ	R_o = 26.33 kΩ

The combined gain produced by the active filter stages at ω = 0 or at infinity is 3.17422, which includes the product of the three K values and either the capacitor or resistor ratio of 0.45269. The voltage divider resistor values can be calculated accordingly. The resulting bandstop filter is shown in Figure 4.12.

$$R_x = 31.74\ \text{k}\Omega, \quad R_y = 14.60\ \text{k}\Omega$$

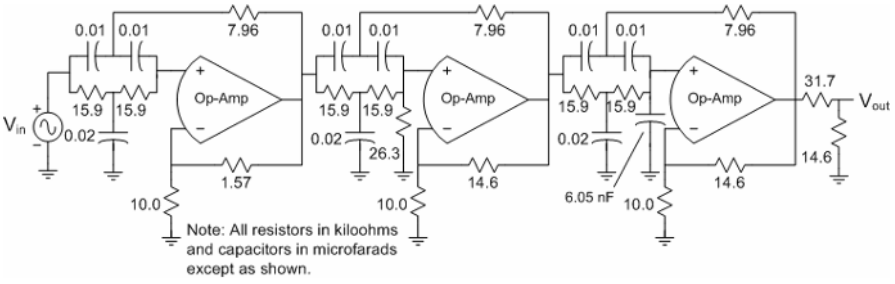

0.01
0.01
7.96
15.9
15.9
0.02
Op-Amp
1.57
10.0
26.3
14.6
6.05 nF
31.7
V_out
V_in
Note: All resistors in kiloohms and capacitors in microfarads except as shown.

Figure 4.12 Chebyshev bandstop active filter for Example 4.4.

4.6 IMPLEMENTING COMPLEX ZEROS WITH ACTIVE FILTERS

Up to this point in the chapter, we have not discussed the implementation of rational functions such as the inverse Chebyshev and elliptic approximations. In previous chapters, we discovered that the primary difference between all-pole and rational approximation functions was that the rational functions required complex zeros on the $j\omega$ axis in the s-plane. The general form of a rational function quadratic factor is shown below. If the a_1 coefficient is zero (as it is for zeros on the $j\omega$ axis), the quadratic factor is identical to the form of a bandstop function derived for the all-pole response in the last section.

$$H_{a,S}(s) = \frac{G \cdot (s^2 + a_1 \cdot s + a_2)}{s^2 + b_1 \cdot s + b_2} \tag{4.59}$$

Therefore, we have already developed an active filter form to implement the inverse Chebyshev and elliptic approximations. It is the twin-tee filter presented in the previous section. We can use that form for any of the selectivities (lowpass, highpass, bandpass, or bandstop) when designing inverse Chebyshev or elliptic filters.

However, there are a few special concerns that must be considered when implementing filter approximation functions with complex zeros. If a lowpass or highpass approximation function has an odd-order, the first-order factor should still be implemented by the appropriate first-order stage discussed previously. In addition, for the case of a bandpass approximation, if the function was derived from an odd-order lowpass function, then the quadratic factor associated with the first-order factor should be implemented by a Sallen-Key bandpass active filter section. The other quadratic factors of the bandpass function and the quadratic factors of the lowpass and highpass functions are then implemented by the twin-tee notch filter. Of course, all stages of a bandstop filter of any approximation use the twin-tee configuration. Two examples will now be used to demonstrate the procedure of implementing inverse Chebyshev and elliptic approximations.

Example 4.5 Inverse Chebyshev Lowpass Active Filter Design

Problem: Determine the resistor and capacitor values to implement an inverse Chebyshev lowpass active filter to meet the following specifications:

$$a_{\text{pass}} = -1 \text{ dB},\ a_{\text{stop}} = -60 \text{ dB},\ f_{\text{pass}} = 100 \text{ Hz, and } f_{\text{stop}} = 300 \text{ Hz}$$

Solution: In this case, a fifth-order filter will be used. We can determine the required approximation function to be

$$H_a(s) = \frac{10.886\cdot 10^{-3}(865.77)\cdot(s^2+3.9282\cdot 10^6)\cdot(s^2+10.284\cdot 10^6)}{(s+865.77)\cdot(s^2+449.34\cdot s+629.45\cdot 10^3)\cdot(s^2+1{,}305.7\cdot s+698.64\cdot 10^3)}$$

The two quadratic terms must be matched to two twin-tee active filter stages while the first-order factor can be implemented by a simple RC stage. By again picking $C = 0.01$ μF and $R_A = 10$ kΩ, the remaining values can be calculated. Note that in this lowpass case, both twin-tee sections employ a capacitor as the additional admittance element.

m	R_m	K_m	R_{Bm}	Y_m
0	115.5 kΩ	—	—	—
1	50.46 kΩ	3.9129	29.13 kΩ	C_o = 26.20 nF
2	31.18 kΩ	5.8634	48.63 kΩ	C_o = 68.60 nF

We determine the gain adjustment factor by finding K_{tot}, as shown below.

$$K_{\text{tot}} = \prod_m K_m = 22.943$$

If the desired output resistance is 10 kΩ, the following values result for the voltage divider with the final filter shown in Figure 4.13.

$$R_x = 229.4 \text{ k}\Omega, \quad R_y = 10.46 \text{ k}\Omega$$

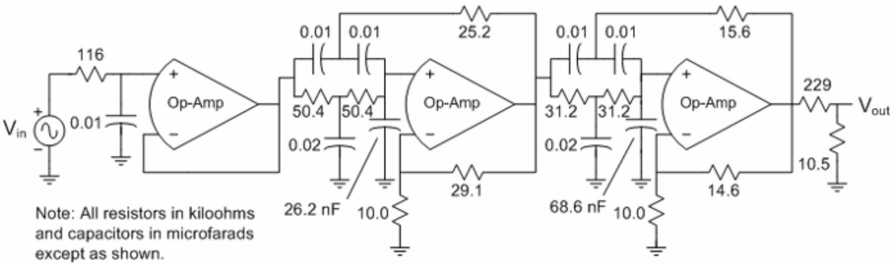


Figure 4.13 Inverse Chebyshev active lowpass filter for Example 4.5.

Example 4.6 Elliptic Bandpass Active Filter Design

Problem: Determine the resistor and capacitor values to implement an elliptic bandpass active filter to meet the following specifications:

$a_{\text{pass}} = -1$ dB,	$a_{\text{stop}} = -60$ dB,	$f_{\text{pass1}} = 250$ Hz,
$f_{\text{pass2}} = 400$ Hz,	$f_{\text{stop1}} = 100$ Hz, and	$f_{\text{stop2}} = 1{,}000$ Hz

Solution: The order of the equivalent lowpass filter is 3, which indicates that a sixth-order bandpass approximation function will be necessary. The transfer function is shown below:

$$H_a(s) = \frac{20.82 \cdot 10^{-3} \cdot 469.8 \cdot (s^2 + 50.07 \cdot 10^6) \cdot (s^2 + 311.3 \cdot 10^3)}{(s^2 + 469.8 \cdot s + 3.948 \cdot 10^6)(s^2 + 178.5s + 2.503 \cdot 10^6)(s^2 + 281.6s + 6.227 \cdot 10^6)}$$

These three quadratic terms must be matched to three active filter stages. The stage related to the first-order factor in the *normalized* LP filter is implemented by the standard Sallen-Key bandpass filter. The other two stages are implemented using the twin-tee filters. By picking C = 0.01 μF and R_A = 10 kΩ, the remaining values can be calculated as shown below. One of the twin-tee stages will have a R_o = 18.86 kΩ and the other will have C_o = 95.04 nF.

m	R_m	K_m	R_{Bm}	Y_m
0	71.18 kΩ	3.6656	26.66 kΩ	—
1	14.13 kΩ	11.251	102.5 kΩ	C_o=95.04 nF
2	179.2 kΩ	11.251	102.5 kΩ	R_o=18.86 kΩ

The combined gain produced by the active filter stages at ω = 0 is 4.4406 and that can be compensated by a voltage divider after the last stage. The voltage divider resistor values are shown below and are calculated to provide an output impedance of 10 kΩ. The resulting bandstop filter is shown in Figure 4.14.

$$R_x = 44.4 \text{ k}\Omega, \quad R_y = 12.9 \text{ k}\Omega$$

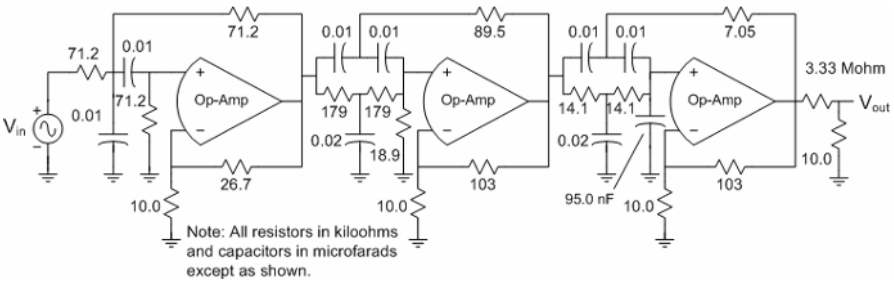


Figure 4.14 Elliptic bandpass active filter for Example 4.6.

4.7 ANALOG FILTER IMPLEMENTATION ISSUES

In the previous sections, we determined component values necessary to implement our analog filter designs in the form of active filters. These active filters make use of electronic components that can be manufactured only to a finite accuracy. The components are not perfect when they are first manufactured and can change value because of aging or exposure to heat, chemicals, or humidity. Therefore, it is important to select the best type of component for our purpose (active filter implementation). Although we can buy precision components that resist changes, the cost of such components can be expensive. It is most cost effective to specify precision components only for those positions that actually require it. We can determine which components are in critical positions by performing a sensitivity analysis.

4.7.1 Component Selection

Each active filter stage is made up of a combination of resistors, capacitors, and op-amps. There are important issues in the selection of each of these components.

We start with a discussion of the active device in our design and then discuss capacitors and resistors in turn.

The op-amp is the amplifying device in our design and the response of the filter is based on the assumption that the op-amp is ideal. We assume that the input impedance is infinite (actually 1 M–100 MΩ depending on device), the output impedance is zero (actually 100–1,000 Ω), and that the op-amp can amplify frequencies up to infinity (actually the bandlimit is usually 1 M–10 MHz). This later characteristic is specified by the op-amp's gain-bandwidth product (GBP). This number provides a reference that can be used to determine the open loop gain of the op-amp at any frequency (or vice versa). For example, the GBP of a 741 op-amp is 10^6, which indicates that this amplifier has an open loop gain of 1,000 at a frequency of 1,000 Hz. Alternatively, we could use this GBP to predict that an upper frequency limit of 20,000 Hz was available with an open loop gain of 50. The important point to remember, however, is that the transfer functions of the active filters were derived assuming that the gains of the op-amps used were very large for all frequencies of interest in the design. Therefore, whether or not the active filter is a lowpass, highpass, bandpass, or bandstop filter, the open loop gain of the op-amp must be large for all frequencies of interest. These frequencies of interest include not only the passbands, but also the stopbands. Luckily, there are many options (other than 741s) when it comes to specifying the active device to be used.

Capacitors represent the reactive elements in our active filter because they have a reactance that changes with frequency (unlike resistors). There are many types of capacitors manufactured today including electrolytic, ceramic, poly, film, and others. Each capacitor type has a place and is manufactured to serve a specific purpose in the electronics world. For example, electrolytic capacitors are generally used in power applications where large values (up to 10,000 uF) are important and variations in value are not the most important criterion for selection. They are the farthest from ideal of any of the capacitor types and should not be used in active filter design. Ceramic capacitors generally have very small values (in the pF range) and are used most often in applications operating in the megahertz range. Generally, active filters will not be operating at that frequency because of the limitation of the op-amp. The most commonly used capacitors for active filters are referred to as "poly" capacitors. There are a number of types that fall into this category (polystyrene, polypropylene, polycarbonate, and polyester), where the different names indicate the dielectric used in their construction. There are trade-offs with regard to tolerance, cost, and characteristics, but in general you will find these types effective in active filter design.

In the case of resistors, there are a variety of types, including carbon composition, carbon film, metal film, wire-wound, and others. Wire-wound resistors are rarely used except in power applications, and carbon composition resistors generate excessive amounts of noise. Therefore, in our application, either metal film or carbon film resistors are usually used. You will find carbon film devices used for 5% tolerance values (meaning that the value of the resistor could be ±5% of its nominal value). The metal film device will have a lower temperature coefficient (meaning that it will display less variation with changes in temperature) and is generally manufactured to 1% tolerance. Values are typically available from 10 Ω to 10 MΩ. Because of the nature of the active devices used (op-amps) it is best to keep the resistor values used in a filter between 1 kΩ and 100 kΩ, if possible, and certainly between 100 Ω and 1 MΩ. Otherwise, the assumptions about the impedances of the op-amp being ideal are no longer valid and the response of the filter may not be as expected.

Hopefully, this brief discussion will get you started in the implementation of your active filter. For further information, consult the references given in the analog filter design section of Appendix A.

4.7.2 Sensitivity Analysis

After selecting the best components for the active filter, it is important to identify which of these components has the most effect on the overall performance of the filter. Since no component is perfect, and they all will change with age, temperature, and other influences, these effects on the filter response can be minimized by selecting the most critical components to have the lowest tolerances to change. We can determine the most critical components in a design by performing a sensitivity analysis. A sensitivity analysis is the process of finding out how any or all of the characteristics of a filter are affected by each and every component that makes up the filter.

For example, the sensitivity of a function *F* with respect to *x* is defined in (4.60). Note that sensitivity not only considers the change in *F* as a function of *x*, but also the nominal values of *F* and *x*. Therefore, the sensitivity considers the per unit change of the function with respect to the per unit change of the parameter. Using the definition of (4.60), other common sensitivity relationships can be determined as shown in (4.61)–(4.66). (The value *c* is considered a constant and *G* is another function of *x*.)

$$S_x^F = \frac{x}{F} \cdot \frac{\partial F}{\partial x} = \frac{\partial[\ln(F)]}{\partial[\ln(x)]} \tag{4.60}$$

$$S_x^{cx} = 1 \tag{4.61}$$

$$S_x^{1/F} = -S_x^F \tag{4.62}$$

$$S_x^{FG} = S_x^F + S_x^G \tag{4.63}$$

$$S_x^{F/G} = S_x^F - S_x^G \tag{4.64}$$

$$S_x^{F^c} = c \cdot S_x^F \tag{4.65}$$

$$S_{x^c}^F = \frac{1}{c} \cdot S_x^F \tag{4.66}$$

In each of the preceding sections, a transfer function for an active filter was given in terms of the resistor and capacitor values used to make up the circuit. In general, those functions can be specified in terms of the pole and zero frequencies and quality factors (*Q*s) as shown in (4.67):

$$H(s) = K \cdot \frac{s^2 + (\omega_z/Q_z) \cdot s + \omega_z^2}{s^2 + (\omega_p/Q_p) \cdot s + \omega_p^2} \tag{4.67}$$

If we consider the lowpass active filter of Section 4.2, which has a transfer function as indicated in (4.68), we can easily match terms to identify the pole frequency and *Q* in terms of the component values as shown in (4.69) and (4.70):

$$H_{c,L}(s) = \frac{K/R_1R_2C_1C_2}{s^2 + [1/R_1C_1 + 1/R_2C_1 + (1-K)/R_2C_2] \cdot s + 1/R_1R_2C_1C_2} \tag{4.68}$$

$$\omega_p = 1/\sqrt{R_1R_2C_1C_2} \tag{4.69}$$

$$Q_p = \frac{1/\sqrt{R_1R_2C_1C_2}}{\dfrac{1}{R_1C_1} + \dfrac{1}{R_2C_1} + \dfrac{1-K}{R_2C_2}} \tag{4.70}$$

The sensitivity of either one of the parameters listed above to any one of the resistor or capacitor values can now be determined. For example,

$$S_{R_1}^{\omega_p} = -0.5 \tag{4.71}$$

which indicates that for every 1.0% *increase* in R_1 there is a 0.5% *decrease* in ω_p. Likewise, we find that

$$S_{R_1}^{Q_p} = -0.5 + Q_p \sqrt{\frac{R_2 C_2}{R_1 C_1}} \tag{4.72}$$

which for equal resistor and capacitor values becomes

$$S_{R_1}^{Q_p} = -0.5 + Q_p \tag{4.73}$$

Equation (4.73) indicates that the sensitivity of Q_p with respect to R_1 can be quite high since Q values can reach 50 to 100. Therefore, the selection of R and C values for the active filters plays an important part in the overall sensitivity of the circuit. We do not have space for a full treatment of sensitivity analysis in this text. However, references in Appendix A provide further details on sensitivity and on methods of selecting component values that yield low sensitivities. The text by Daryanani provides a particularly good presentation. In fact, that text presents a different choice for the component values for a Sallen-Key active filter as listed below. By making those selections, it can be shown by substitution into (4.70) that Q_p is no longer a function of R at all and therefore would have a sensitivity of zero with respect to changes in resistance value:

$$K = 1, \quad \mathrm{R}_1 = R_2 = R$$

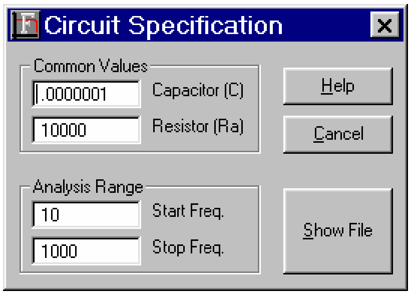
$$C_1 = \frac{2 \cdot Q_p}{R \cdot \omega_p}, \quad \mathrm{C}_2 = \frac{1}{2 \cdot R \cdot Q_p \cdot \omega_p}$$

A sensitivity analysis provides us with valuable information about the component values used in the circuit. With that information, we can determine which components are critical in controlling variation in key filter parameters. However, it is also very important to perform a worst-case analysis on the circuit. This analysis of the circuit would use the sensitivities that have already been determined to set each component to the extreme limit of its tolerance in order to see the effect on the overall circuit performance. For example, all components with negative sensitivities would be set to their lower tolerance limit while all components with positive sensitivities would be set to their upper limits. The circuit would then be analyzed to see if it still met the specifications. Next, all component values would be set to the opposite limit and the test would be repeated. If a circuit passed this test, it should perform up to specifications with the randomly chosen values used when it is assembled. An example of a worst-case analysis using PSpice is given in Section 4.8.

Another type of statistical test that is less rigorous than a worst-case test, but probably more typical of what will happen in real life, is the Monte Carlo simulation. In this test, component values are selected randomly within their tolerance range and the circuit is tested to see if it meets the required specifications. In most software packages, the values can be characterized as having a Gaussian or uniform distribution. A number of Monte Carlo tests are usually run to simulate the variation of component values that will be seen in the normal assembly process.

4.8 USING WFILTER IN ANALOG FILTER IMPLEMENTATION

In order to see how WFilter helps in analog filter implementation, we generate the component values and PSpice data file for the problem given in Example 4.5. After we enter the filter specifications and design the filter, we can specify the common component values and frequency specification in the ***Circuit Specification*** dialog box shown in Figure 4.15. This dialog box is displayed by selecting ***Options*** and then ***Generate Spice File*** from the menu bar.


Circuit Specification

Common Values

.0000001 Capacitor (C)

10000 Resistor (Ra)

Analysis Range

10 Start Freq.

1000 Stop Freq.

Figure 4.15 Circuit specification dialog box.

We have entered a common capacitor value of 0.1 μF and common resistor value of 10 kΩ. In addition, we have specified the frequency analysis range to be from 10 Hz to 1 kHz. After pressing the ***Show File*** button, the PSpice text file shown in Listing 4.1 is displayed. Notice that the first stage is first-order with an R_B value of one ohm (most circuit analysis tools do not accept zero ohms), while the other two stages describe twin-tee notch filters. The original calculated values are shown in the listing and should be used in a test analysis of the circuit to determine if the specifications have been met. If the specifications are not met with these "ideal" values, either the op-amp model is not adequate or an error has been encountered in the design steps.

```
Example 4.5 and PSpice Example
*    Specify input source:
Vs   11   0  AC   1   0
*     Stage Number   1            (Practical Values)
R11  11  12  1.155E+04            (1.15E+04)
C11  12   0  1.000E-07            (1.00E-07)
Rb1  13  21  1.000E+00            (1.00E+00)
X1   12  13  21  OPAMP
*    Stage Number   2
C21  21  22  1.000E-07            (1.00E-07)
C22  22  24  1.000E-07            (1.00E-07)
C23  23   0  2.000E-07            (2.00E-07)
R21  21  23  5.046E+03            (4.99E+03)
R22  23  24  5.046E+03            (4.99E+03)
R23  22  31  2.523E+03            (2.49E+03)
Ra2  25   0  1.000E+04            (1.00E+04)
Rb2  25  31  2.913E+04            (2.94E+04)
Co2  24   0  2.620E-07            (2.70E-07)
X2   24  25  31  OPAMP
*    Stage Number   3
C31  31  32  1.000E-07            (1.00E-07)
C32  32  34  1.000E-07            (1.00E-07)
C33  33   0  2.000E-07            (2.00E-07)
R31  31  33  3.118E+03            (3.09E+03)
R32  33  34  3.118E+03            (3.09E+03)
R33  32  41  1.559E+03            (1.54E+03)
Ra3  35   0  1.000E+04            (1.00E+04)
Rb3  35  41  4.863E+04            (4.87E+04)
Co3  34   0  6.860E-07            (6.80E-07)
X3   34  35  41  OPAMP
*     Voltage divider section
Rx   41  42  2.294E+05            (2.32E+05)
Ry   42   0  1.046E+04            (1.05E+04)
*     Sub-circuit model for op-amp
.SUBCKT OPAMP  1  2  6
Ri    1   2   1.000E+08
E1    3   0   1   2  1.000E+03
Rx    3   4   1.000E+03
Cx    4   0   1.000E-09
E2    5   0   4   0  1.000E+03
Ro    5   6   1.000E+00
.ENDS
*     Analysis modes
.AC   DEC    100     1.000E+01  1.000E+03
.PROBE
.END
```

Listing 4.1 Circuit analysis data file for Example 4.5.

The next step in the testing of the active filter is to replace the ideal components with practical values. For this exercise, we will use 1% resistor and 2% capacitor values. (The selected values are shown in italics and parentheses in the listing.) A worst-case analysis can then be run on the circuit that involves adding tolerance information for the component values. (You need access to the PSpice program to make this run.) During the worst-case analysis, sensitivities for each component are determined, and as the final step, each component is set to its worst-case extreme. Figure 4.16 shows both the worst-case and nominal response.

If the two frequency responses were carefully compared, we would see that there is a +2.5 dB “bump” in the passband, and we lose over 2 dB in the stopband attenuation. These changes should represent the “worst” that can happen due to the unfortunate selection of the worst possible combination of components. (A worst-case analysis was also run with 5% resistor values and 10% capacitor values and the “bump” increased to 15 dB.) A Monte Carlo analysis can also be run on the circuit to indicate the more typical extremes that might be encountered.

Depending on the nature of the application, we can live with the resulting variations, select even more precise (expensive) components, or redesign the filter to more stringent specifications than actually desired. This redesigned filter would then be able to vary to some degree while still satisfying the real specifications. However, this filter may also require a higher order, which will add cost to the project.

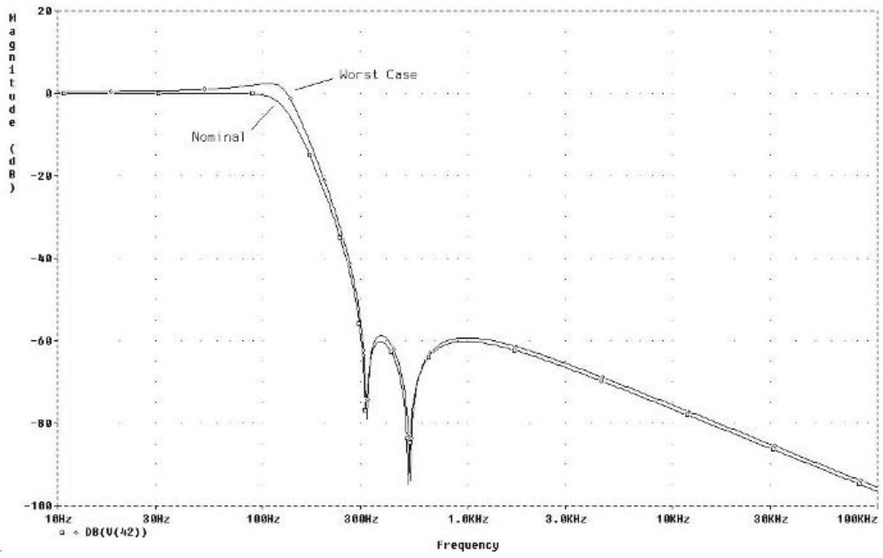


Figure 4.16 Frequency responses for Example 4.5.

4.9 CONCLUSION

We have reached the end of the first part of this text. We were able to design a variety of analog filters, view their frequency responses, and implement them in an active filter form. Of course, we have left a good deal of material uncovered. There are other filter types that could have been discussed. There are other features that could have been included in the frequency response calculation and display. Moreover, there are other implementation techniques available for analog filters. However, it is now time to move into the realm of digital filters. We will see that our work in the area of analog filters will prove valuable, since one form of a digital filter uses much of what we have learned so far. For those who are interested in seeing the code used to generate the PSpice circuit file, please turn to Appendix F.

Chapter 5

Introduction to Discrete-Time Systems

Although most naturally occurring signals are of the analog variety (continuous-amplitude and continuous-time variation), we are finding that conversion of these signals to a digital form (discrete-time and discrete-amplitude variations) provides many advantages. For example, digital signals can be stored on computer floppy or hard disks. They can be compressed to save space, converted to other formats, or transmitted in combination with other signals. Digital forms of signals are truly becoming the standard in everyday use as compact discs (CDs) for audio and multimedia applications can attest. Therefore, the remainder of this text is devoted to the application of filtering techniques to digital signals.

The material in this chapter should provide a review of the basic principles of discrete-time systems that are necessary to understand the material presented in the remainder of the text. In the first section of this chapter we will discuss the process of converting analog signals into a digital form. Next, we will develop methods of dealing with discrete-time signals in both the time domain and frequency domain. We will also learn how to find the frequency response of a discrete-time system as well as how to play digitized waveforms on a computer equipped with a sound card.

5.1 ANALOG-TO-DIGITAL CONVERSION

As indicated in the introduction, most of the signals that we deal with every day are known as analog signals. This type of signal has a continuous variation in both time and amplitude, as shown in Figure 5.1(a). In order to convert this analog signal to a digital signal with discrete-time and discrete-amplitude, as shown in Figure 5.1(b), several steps must be performed.

First, the frequency spectrum of the analog signal must be strictly band-limited. Second, the signal must be sampled at the proper sampling rate. Third, the sampled value must be quantized to an acceptable level of accuracy. When it is time to convert the digital signal back to analog form, there are a number of methods that can be used, but one simple method requires only two steps. The first step is to output the digital value of the signal and hold it for the duration of the sample period. The second step is to pass that signal through a lowpass filter. In order to understand the reasons why these steps are necessary for analog and digital conversion, we need to study the frequency spectrum of a sampled signal and the requirements placed on the sampling rate.

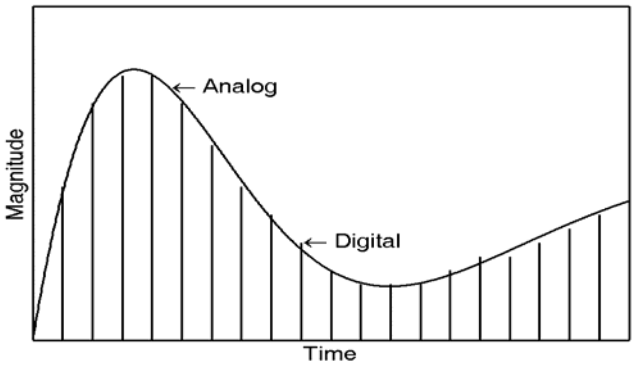


Figure 5.1 Comparison of analog and digital signals.

5.1.1 Frequency Spectrum and Sampling Rate

When an analog signal $x(t)$ is sampled, as shown in Figure 5.1, the samples are usually taken at equal intervals of time. This sampling period T_S is the inverse of the sampling frequency f_S. The resulting digitized waveform $x_S(nT)$ can be specified with an argument indicating the sampling period T_S, as shown in (5.1):

$$x_s(nT) = x(t)\big|_{t=nT_s} \tag{5.1}$$

For example, if we had an analog signal $x(t)$ as specified in (5.2), the sampled version of the signal $x_S(nT_S)$ as shown in (5.3) would result:

$$x(t) = 50 \cdot e^{-100 \cdot t} \cdot \cos(200 \cdot t) \tag{5.2}$$

$$x(nT) = 50 \cdot e^{-100 \cdot nT_s} \cdot \cos(200 \cdot nT_s) \tag{5.3}$$

If we assume that the analog signal is sampled at a frequency of 1,000 samples per second, then we can calculate the values of $x_S(nT_S)$ with $T_S = 0.001$ second. The values that result can be stored as a sequence of numbers, as shown below. This is an important point: once an analog signal has been digitized, it is nothing more than a sequence of numbers that can be stored, manipulated, transmitted, or processed in any way we see fit.

$$x(nT_s) = \{50.0, 44.3, 37.7, 30.6, 23.4, 16.4, ...\}, n = 0, 1, \ldots \tag{5.4}$$

Since the sampling period seldom changes, discrete-time equations usually drop the sampling period T_S from the expressions to produce an expression such as (5.5). The sampling period will not be needed again until the digitized waveform is converted back to analog form.

$$x(n) = 25 \cdot e^{-10 \cdot n} \cdot \cos(100 \cdot n) \tag{5.5}$$

Although it doesn't appear that much has changed in the representation of the signal in the time domain, a great deal has changed in the frequency domain. The frequency spectrum of a signal is shown in Figure 5.2(a) before sampling. If this signal were sampled at a frequency of f_S, the spectrum of the sampled signal would be as shown in Figure 5.2(b). The original analog spectrum is replicated throughout the spectrum at intervals of f_S (although only one instance of that is shown). Because of this replication, it is feasible that there will be corruption of the frequency components of the original signal by components of the replicated signals. This corruption is referred to as *aliasing*, and the offending frequencies are *alias* frequencies. (Further details of aliasing and the upcoming Nyquist criteria can be found in most of the digital filter design references in Appendix A.)

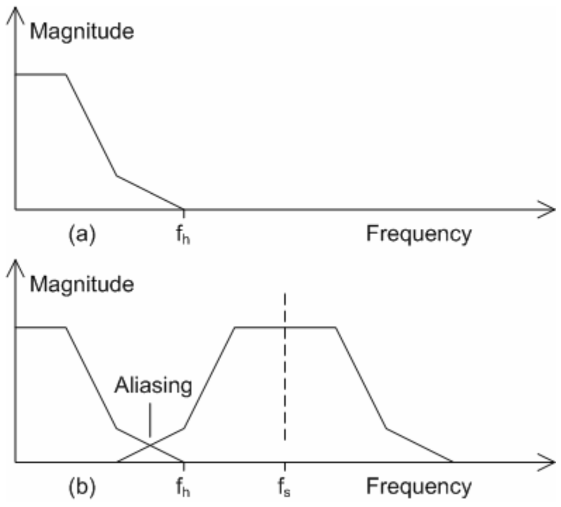


Figure 5.2 Spectrum of signal (a) before and (b) after sampling.

When digitizing a signal, it is very important to capture most, if not all, of the information present in the original analog signal without generating alias frequencies. For this reason, a good deal of study has gone into the conditions necessary to faithfully convert an analog signal into a digital form. We can see by closely observing Figure 5.2(b) that if we are to eliminate the effect of aliasing, the sampling frequency must be at least twice as high as the highest frequency in the original signal. This relationship, as shown in (5.6), is known as the Nyquist criteria, and is a well-known requirement in sampling theory:

$$f_s > 2 \cdot f_h \tag{5.6}$$

In order to guarantee that this requirement is met at all times, it is normal procedure to band-limit the input signal to one-half of the sampling frequency once the sampling frequency has been set. This is a prudent measure, since frequencies beyond those which are normally expected can occasionally occur in all systems. The band-limiting process can be implemented by sending the analog signal through an analog lowpass filter prior to sampling. (What an excellent use of our analog filter theory!)

5.1.2 Quantization of Samples

Once the analog signal has been sampled, it is a discrete-time signal since values of the signal exist only at particular moments of time, but the amplitude of the signal is still continuous. Then, the next step in the analog-to-digital conversion (ADC) is to quantize the continuous-amplitude signal to one of many discrete values of amplitude. The number of possible values allowed for the amplitude is determined by the size of the variable chosen to store the values. For example, if a single byte of memory (8 bits) is chosen to store the information, then the amplitude can take on one of 2^8 or 256 different values. If the original signal had a range of amplitudes from +1 volt to –1 volt, then the difference between adjacent amplitudes would be approximately $7.8 \cdot 10^{-3}$ volts. On the other hand, if two bytes of memory (16 bits) were used to store each sample, there would be 2^{16} or 65,536 different values to represent the signal. With this many values, the ±1 volt signal would have adjacent amplitudes separated by only $3.05 \cdot 10^{-5}$ volts. Obviously, the larger the variable used to store the sampled data, the more closely we can approximate the analog signal with the digital representation. (In the previous discussion it is assumed that uniform sampling was used where all levels would be equally spaced. There are also techniques that use nonuniform spacing to place more levels at lower levels and wider spacing for larger signals.)

However, the drawback of using larger and larger variables is twofold. First, the storage requirements to store the digitized waveform are proportional to the number of bits used to quantize the samples. For example, suppose we decide to sample a speech signal that contains signal frequencies from 300 to 3,000 Hz at a frequency of 8,000 Hz (which satisfies Nyquist's criteria). The waveform would need a file size of 480,000 bytes (480 kilobytes) to store one minute's worth of data using only 1 byte per sample. On the other hand, if we stored one minute of stereo music using 2 bytes of data for each channel (left and right), the file size would need to be over 10 million bytes (10 megabytes). This large file size is necessary to accommodate a sampling rate of 44 kHz, which is the normal rate used for high-fidelity audio signal with frequencies up to 20 kHz.

The second drawback is the speed of conversion from analog-to-digital form. Although ADC chips are very fast these days, obtaining more accuracy requires more time for conversion. Eventually, a limit on accuracy will be reached because the conversion cannot be made in the allotted sample interval. This limitation is more common when processing video signals that have bandwidths in the millions of hertz (MHz).

It is important to note at this point that theoretically the sampling of an analog waveform does not normally produce any error. (This assumes that the sample clock does not introduce error because of timing jitter.) It is the quantization of the sample that produces the error in a digital system. If the samples could be stored in their original continuous-amplitude form, they could be used to regenerate the original signal with no error (assuming the Nyquist criteria is met). The maximum amount of error introduced into the system by this quantization is equal to one-half of the difference between amplitude levels. As we can see, selecting a method for the digitization of an analog signal is a compromise between conversion speed, waveform accuracy, and storage or transmission size.

5.1.3 A Complete Analog-to-Digital-to-Analog System

Figure 5.3 shows a block diagram of a complete system that first converts an analog signal to digital form for processing, transmission, or storage. Then the conversion is undone by converting the digital signal back to analog form at another time and/or place using a digital-to-analog converter (DAC). As shown in Figure 5.2(b), the process of sampling a signal produces replicas of the original analog spectrum at intervals of the sampling frequency. In order to recover the original analog signal we simply have to eliminate the frequencies higher than $f_S/2$. This filtering is accomplished by a high-order lowpass filter.

A good example of such a complete procedure is the processing of audio signals for a music CD. The original sounds of the music are supplied by instruments or voices and are then recorded on tape in analog form. At some later time, these analog signals are digitized and encoded on the compact disc. We can then buy this CD and take it home and play it on our stereo system where the musical data is first converted from digital to analog form and then reproduced for us. In this example, the data is processed, stored, and reproduced at a later time and different place.

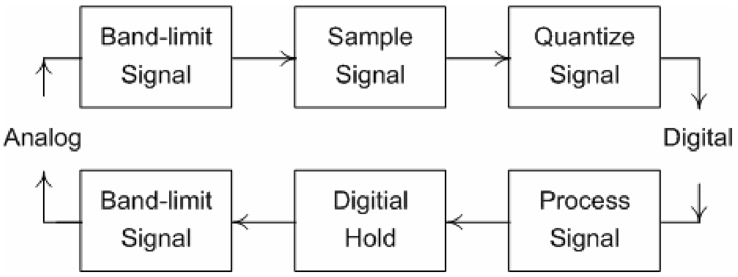


Figure 5.3 Complete analog-to-digital-to-analog system.

5.2 LINEAR DIFFERENCE EQUATIONS AND CONVOLUTION

In our study of discrete-time systems, we need to be able to describe them in a number of different ways. In this section we will learn how to describe a discrete-time system by using a difference equation and the system's impulse response. The impulse response of a system is simply its response to the discrete impulse function $\delta(n)$, as defined in (5.7). This function is the discrete-time equivalent to the Dirac delta function $\delta(t)$ used in continuous-time system theory. The impulse function is a very useful function because any input signal sequence can be described by summing weighed and delayed versions of the impulse function. Likewise, a discrete-time system's response can be described by combining the responses to the input sequence.

$$\delta(n) = \begin{cases} 1, \text{for } n = 0 \\ 0, \text{for } n \neq 0 \end{cases} \tag{5.7}$$

In addition, we will use the unit step function $u(n)$ in many of our expressions, so its definition is shown in (5.8). This function is the equivalent of the continuous-time step function $u(t)$:

$$u(n) = \begin{cases} 1, \text{for } n \geq 0 \\ 0, \text{for } n < 0 \end{cases} \tag{5.8}$$

Just as $u(t)$ can be defined as the integral of $\delta(t)$, $u(n)$ and $\delta(n)$ have a relationship built on the infinite summation as described in (5.9):

$$u(n) = \sum_{n=0}^{\infty} \delta(n) \tag{5.9}$$

5.2.1 Linear Difference Equations

Continuous-time systems often require the solution of one or more linear differential equations. In this case, values of the input and output signals and their derivatives are used. The discrete-time equivalent to this analysis uses linear difference equations that make use of past and present values of the input and past values of the output. Note that instead of using derivatives, we are using past values of the signals.

One classic example of a discrete-time system that can be easily described by a difference equation is the bank account balance. Let $y(n)$ reflect the balance in a savings account where $x(n)$ dollars are deposited at the beginning of each month. We can assume that the account earns interest at a monthly percentage rate of I. If we let T_S, the sample period of the system, be one month, then the difference equation shown in (5.10) reflects the balance in the account at the beginning of each month:

$$y(n) = y(n-1) + (I/100) \cdot y(n-1) + x(n) \tag{5.10}$$

If we assume that we deposit \$100 each month and the interest is 1% per month, we can replace $x(n)$ with $100 \cdot u(n)$ in (5.10) to produce the following result:

$$y(n) = (1.01) \cdot y(n-1) + 100 \cdot u(n) \tag{5.11}$$

Table 5.1 shows the balance of the account for six months, assuming that we just opened the account and the balance was zero.

Table 5.1

Balance in Savings Account

Month	*Balance*
0	\$100.00
1	\$201.00
2	\$303.01
3	\$406.04
4	\$510.10
5	\$615.20

Although manual techniques for the solution of this difference equation are acceptable for six months' time, other methods will need to be employed for longer periods of time. Let’s see if we can determine a general formula for the balance of this account. Starting with the first month we find that

$$y(0) = 100 \tag{5.12}$$

$$y(1) = 1.01 \cdot y(0) + 100 = 100 \cdot (1 + 1.01) \tag{5.13}$$

$$y(2) = 1.01 \cdot y(1) + 100 = 100 \cdot (1 + 1.01 + 1.01^2) \tag{5.14}$$

or in general

$$y(n) = 100 \cdot \sum_{k=0}^{n} 1.01^k \tag{5.15}$$

Although this expression is compact, it still would require the calculation of the sum of n + 1 terms in order to determine the value. What we really need is a closed form solution. (Of course I wouldn’t have mentioned such a thing if one didn’t exist.) We can define what is referred to as a finite geometric sum, as shown below:

$$\text{FGS} = \sum_{k=0}^{n} a^k \tag{5.16}$$

Then, with some ingenious mathematics, we can define a difference that cancels most of the terms:

$$\text{FGS} - a \cdot \text{FGS} = \sum_{k=0}^{n} a^k - a \cdot \sum_{k=0}^{n} a^k = 1 - a^{n+1} \tag{5.17}$$

Finally, we can determine the value of the FGS, as shown below. (The value of FGS when a = 1 is determined directly from (5.16):

$$\text{FGS} = \begin{cases} \dfrac{1 - a^{n+1}}{1 - a}, & \text{for } a \neq 1 \\ n + 1, & \text{for } a = 1 \end{cases} \tag{5.18}$$

We can now write the equation for the balance of our savings account in a closed form, as shown below. This expression does not need the calculation of $n + 1$ terms in order for us to evaluate it.

$$y(n) = 100 \cdot \frac{1 - 1.01^{n+1}}{1 - 1.01} \tag{5.19}$$

We can also define the value of the infinite geometric sum, as shown below:

$$\text{IGS} = \sum_{k=0}^{\infty} a^k \tag{5.20}$$

By simply allowing n in (5.18) to approach infinity, we can see that IGS can be specified as

$$\text{IGS} = \begin{cases} \dfrac{1}{1-a}, & \text{for } |a| < 1 \\ \text{undefined} & \text{otherwise} \end{cases} \tag{5.21}$$

One way to completely define a discrete-time system is by using its difference equation. In general, the difference equation describing the output for a discrete-time system can be written as

$$y(n) = \sum_{k=0}^{M} a_k \cdot x(n-k) - \sum_{k=1}^{N} b_k \cdot y(n-k) \tag{5.22}$$

We would have complete knowledge of the system if we know the coefficients a_k and b_k. As indicated in (5.22), the output $y(n)$ is a function of past and present values of the input $x(n)$ and past values of the output. A system such as this is a *recursive* system. A system that has its output described only by past and present values of the input is a *nonrecursive* system. We will see in the chapters to come that these definitions effectively divide the types of digital filters to be designed into two groups as well.

Notice that we have defined the output of our system in terms of only past and present values of the input. Such a system is referred to as a *causal* system. Any real-time system, of course, must be causal since we cannot determine the output of a system based on input or output values we have not yet seen. However, systems that are not real-time can be *noncausal*. For example, any system that can draw its input from stored data can determine the output of the system at time n by using input values at $n + m$. These “future” values are known since they already have been stored.

5.2.2 Impulse Response and Convolution

Another way to completely describe a discrete system is to specify the impulse response $h(n)$ of the system. The impulse response of a system can be determined from the difference equation by substituting the impulse function $\delta(n)$ for the input $x(n)$ and determining the output $y(n)$.

Example 5.1 Determination of Impulse Response

Problem: Assume that a discrete-time system is described by the difference equation shown below. Determine the first five values of the impulse response, and then formulate an analytic expression for the impulse response.

$$y(n) = b_1 \cdot y(n-1) + x(n)$$

Solution: First, the equation is modified to reflect the fact that the output will be the impulse response $h(n)$ if the input is $\delta(n)$.

$$h(n) = b_1 \cdot h(n-1) + \delta(n)$$

Next, the set of the first five values of the impulse response are determined using Table 5.2.

Table 5.2
Results of Impulse Response

n	$\delta(n)$	$h(n-1)$	$h(n)$
0	1	0	1
1	0	1	b_1
2	0	b_1	$b_1{}^2$
3	0	$b_1{}^2$	$b_1{}^3$
4	0	$b_1{}^3$	$b_1{}^4$

From these results we can see that there is a general form to the impulse response, as shown below:

$$h(n) = b_1^n \cdot u(n)$$

When the impulse response of a system is known, the complete characteristics of the system are known. The reaction of the system to any other input can then be determined by using convolution (similar to the operation in continuous systems). The output of the system can be defined as in (5.23) where the $*$ indicates convolution, not multiplication. In that expression, we see that the output of the system $y(n)$ is determined by computing a sum of products of the impulse response coefficients $h(n)$ and past values of the input $x(n-k)$. Convolution has the commutative property so the order of the functions in the convolution definition is not important. From a practical standpoint however, the simpler function is typically specified in the time-shifted format. This is the predominant method used to implement an FIR filter, which is discussed in detail in Section 8.3. In addition, a number of the texts listed in the digital filter design section of Appendix A cover discrete-time convolution.

$$y(n) = h(n) * x(n) = \sum_{k=-\infty}^{\infty} h(k) \cdot x(n-k) \tag{5.23}$$

Example 5.2 System Response by Convolution

Problem: Determine the output of the system described in Example 5.1 if the input is the signal shown below:

$$x(n) = a_1^n \cdot u(n)$$

Solution: Since we already have the impulse response of the system, we can use convolution to determine the output of the system. The equation for the output of the system is

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

or

$$y(n) = \sum_{k=-\infty}^{\infty} a_1^k \cdot u(k) \cdot b_1^{(n-k)} \cdot u(n-k) = b_1^n \cdot \sum_{k=0}^{n} (a_1/b_1)^k \cdot u(n)$$

Note that the limits on the convolution summation are adjusted by the functions $u(k)$ and $u(n-k)$. The step function $u(k)$ has zero value for negative values of k and therefore the lower limit of the summation is set to zero. The step function $u(n-k)$ will have zero value for all k values greater than n, and therefore the upper limit of the summation is set to n. We can use the form of a finite geometric sum to simplify the result into a closed form solution, as shown below:

$$y(n) = \begin{cases} \dfrac{b_1^{n+1} - a_1^{n+1}}{b_1 - a_1}, \text{for } b_1 \neq a_1 \\ (n+1) \cdot b_1^n, \quad \text{for } b_1 = a_1 \end{cases}$$

5.3 DISCRETE-TIME SYSTEMS AND *Z*-TRANSFORMS

It is also important to be able to understand a discrete-time system's characteristics in the frequency domain as well as the time domain. For linear systems, the Laplace transform can be used to transform time domain characteristics to the frequency domain. For discrete-time systems, we will use the *z*-transform as defined in (5.24):

$$z\{x(n)\} = X(z) = \sum_{n=-\infty}^{\infty} x(n) \cdot z^{-n} \tag{5.24}$$

The *z*-transform of a weighted impulse function, for example, results in a single term because the impulse function has only one nonzero value.

$$z\{A \cdot \delta(n)\} = A \sum_{n=-\infty}^{\infty} \delta(n) \cdot z^{-n} = A \cdot 1 \cdot z^0 = A \tag{5.25}$$

The *z*-transform of a weighted step function can also be determined using the definition. In this case, the result can be simplified by using the definition of the infinite geometric sum. Some of the more common *z*-transform pairs are shown in Table 5.3.

$$z\{A \cdot u(n)\} = A \sum_{n=-\infty}^{\infty} u(n) \cdot z^{-n} = A \cdot \sum_{n=0}^{\infty} \cdot (z^{-1})^n = \frac{A}{1 - z^{-1}} = \frac{A \cdot z}{z - 1} \tag{5.26}$$

The *z*-transform also has a set of useful properties, as shown in Table 5.4. First and foremost is the property that the *z*-transform of the impulse response is the system's transfer function in the *z*-domain. We'll use that property in the next example. The second property in Table 5.4 shows that convolution in the time domain can be represented as simple multiplication in the *z*-domain. The third property listed shows that time delay of *k* units of time can be represented by multiplication by z^{-k} in the *z*-domain. And finally, multiplication by *n* in the time domain can be represented by differentiation in the *z*-domain. For further discussion of these and other properties of the *z*-transform or for a more comprehensive list of *z*-transform pairs, please refer to one of the reference texts listed in Appendix A.

Table 5.3

Common *z*-Transform Pairs

Time Domain Function	*Frequency Domain Function*
$A \cdot \delta(n)$	A
$A \cdot u(n)$	$\frac{A \cdot z}{(z-1)}$
$A \cdot n \cdot u(n)$	$\frac{A \cdot z}{(z-1)^2}$
$A \cdot a^n \cdot u(n)$	$\frac{A \cdot z}{(z-a)}$
$A \cdot \cos(\Omega \cdot n) \cdot u(n)$	$\frac{A \cdot z \cdot [z - \cos(\Omega)]}{z^2 - 2 \cdot \cos(\Omega) \cdot z + 1}$
$A \cdot \sin(\Omega \cdot n) \cdot u(n)$	$\frac{A \cdot z \cdot \sin(\Omega)}{z^2 - 2 \cdot \cos(\Omega) \cdot z + 1}$
$A \cdot a^n \cdot \cos(\Omega \cdot n + \phi) \cdot u(n)$	$\frac{A \cdot z \cdot [z \cdot \cos(\phi) - a \cdot \cos(\phi - \Omega)]}{z^2 - 2 \cdot a \cdot \cos(\Omega) \cdot z + a^2}$

Table 5.4

Common *z*-Transform Properties

Time Domain Function	*Frequency Domain Function*
$h(n)$	$H(z)$
$\sum_{k=-\infty}^{\infty} x_1(k) \cdot x_2(n-k)$	$X_1(z) \cdot X_2(z)$
$x(n-k)$	$z^{-k} \cdot X(z)$
$n \cdot x(n)$	$-z \cdot \frac{dF(z)}{dz}$

Example 5.3 Determining the Transfer Function

Problem: Determine the transfer function of the system described in Example 5.1, which has an impulse response of

$$h(n) = b_1^n \cdot u(n)$$

Find the location of the poles and zeros of the transfer function as well as the difference equation of the system from the transfer function.

Solution: Using the fourth entry in the table of *z*-transform pairs, we can determine that the transfer function of the system described is

$$H(z) = \frac{z}{(z - b_1)} = \frac{1}{(1 - b_1 \cdot z^{-1})} = \frac{Y(z)}{X(z)}$$

We see that there is a single pole (denominator root) at $z = b_1$ and a single zero (numerator root) at $z = 0$. By cross-multiplication, the following equation results:

$$Y(z) - b_1 \cdot z^{-1} \cdot Y(z) = X(z)$$

By taking the inverse *z*-transform of this equation and applying the time shift property of the *z*-transform, we can determine the same difference equation, as in Example 5.1:

$$y(n) = b_1 \cdot y(n-1) + x(n)$$

Another way that a discrete-time system can be described is by drawing a system diagram to represent a general difference equation. Figure 5.4 shows the system diagram for (5.21). In the figure, delays are represented by z^{-1} and multiplication by triangular symbols. Of course, if a nonrecursive filter was being represented, the lower half of the system diagram would be eliminated since the output of such a system does not depend on past values of the output.

In this system diagram, we see that each time the signal moves through a delay element the output is delayed by one sample interval. Although the system diagram is correct and will produce the correct output relationship, there is a more efficient description of the system. First, if we transform the general difference equation of (5.22), we have

$$Y(z) \cdot \left(1 - \sum_{k=1}^{N} b_k \cdot z^{-k}\right) = X(z) \cdot \sum_{k=0}^{M} a_k \cdot z^{-k} \tag{5.27}$$

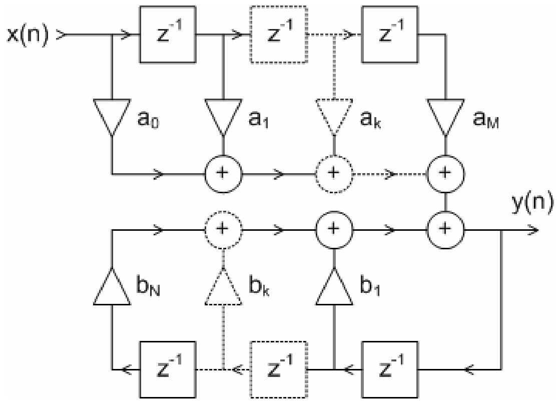


Figure 5.4 System diagram of general discrete-time system.

From this description, we can determine the transfer function $H(z)$ to be

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} a_k \cdot z^{-k}}{1 - \sum_{k=1}^{N} b_k \cdot z^{-k}} \tag{5.28}$$

We can represent (5.28) in a slightly different way to allow for further development:

$$\frac{Y(z)}{W(z)} \cdot \frac{W(z)}{X(z)} = \frac{\sum_{k=0}^{M} a_k \cdot z^{-k}}{1} \cdot \frac{1}{1 - \sum_{k=1}^{N} b_k \cdot z^{-k}} \tag{5.29}$$

This representation leads to two separate relationships, as shown below:

$$Y(z) = W(z) \cdot \sum_{k=0}^{M} a_k \cdot z^{-k} \tag{5.30}$$

$$W(z) = X(z) + W(z) \cdot \sum_{k=1}^{N} b_k \cdot z^{-k} \tag{5.31}$$

These two equations in the z-domain can be written in their equivalent form in the time domain by recognizing that z^{-k} in the z-domain represents a delay of k sample periods in the time domain.

$$y(n) = \sum_{k=0}^{M} a_k \cdot w(n-k) \tag{5.32}$$

$$w(n) = x(n) + \sum_{k=1}^{M} b_k \cdot w(n-k) \tag{5.33}$$

The result of this derivation is that the general form of a discrete-time system diagram can be drawn with more efficient use of the delay units by defining $w(n)$ in the diagram. Since these delay units must be implemented in hardware or software, the fewer used the better. Figure 5.5 shows the preferred method of drawing the system diagram with fewer delays.

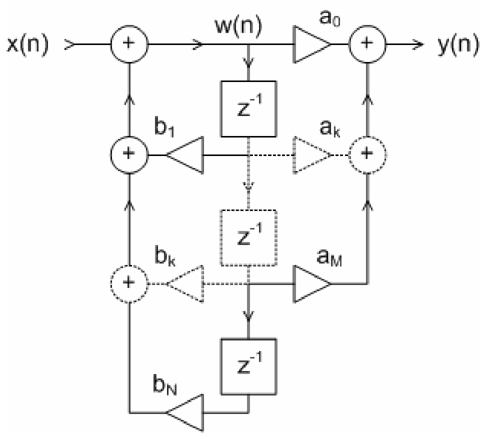


Figure 5.5 System diagram with fewer delay units.

5.4 FREQUENCY RESPONSE OF DISCRETE-TIME SYSTEMS

The frequency response is one of the most important characteristics of a discrete-time system. Although it does not completely describe the system as the difference equation, impulse response, or transfer function does (it does not convey the transient behavior of the system), the frequency response does provide important information about the steady-state behavior of the system. We can begin by considering an analog sinusoidal signal and the sampled signal using a sampling period of T_S.

$$x(t) = A \cdot \cos(\omega \cdot t) \tag{5.34}$$

$$x(nT) = A \cdot \cos(\omega \cdot nT_s) = A \cdot \cos(\Omega \cdot n) \tag{5.35}$$

where we have defined

$$\Omega = \omega \cdot T_s = 2\pi\ f \cdot T_s = 2\pi\ f \,/\, f_s \tag{5.36}$$

Equation (5.36) provides us with a method of comparing the analog frequency f to the digital frequency Ω. The range of analog frequencies acceptable for a discrete-time system does not extend from zero to infinity as would be the case in a normal analog system. We must remember that the upper limit of acceptable analog frequencies in discrete-time systems is governed by the Nyquist criteria, and therefore the range is defined as shown below. Note that a subscript of d is appended to the frequency variable to indicate we are talking about an equivalent analog frequency within a discrete-time system.

$$0 \le f_d < f_s / 2 \tag{5.37}$$

Combining this condition with (5.36) results in the following range of digital frequencies

$$0 \le \Omega < \pi \tag{5.38}$$

Although a sine or cosine function is usually used to determine the frequency response of a hardware system, both of these functions can be described by complex exponentials, as shown below:

$$\cos(\Omega \cdot n) = \frac{e^{j\Omega n} + e^{-j\Omega n}}{2} \tag{5.39}$$

$$\sin(\Omega \cdot n) = \frac{e^{j\Omega n} - e^{-j\Omega n}}{2j} \tag{5.40}$$

In fact, all periodic functions can be represented by these exponentials and even constants can be represented as exponentials with zero frequency. Therefore, when determining the frequency response of a discrete system, it is common to consider the driving function as the complex exponential, as shown below:

$$x(n) = e^{j\Omega n} \tag{5.41}$$

The output of a discrete-time system can then be determined by convolving this input signal with the system's impulse response. The output is then

$$y(n) = \sum_{k=-\infty}^{\infty} h(k) \cdot e^{j\Omega(n-k)} = e^{j\Omega n} \sum_{k=-\infty}^{\infty} h(k) \cdot e^{-j\Omega k} \tag{5.42}$$

Or, we can rewrite the result as

$$y(n) = x(n) \sum_{k=-\infty}^{\infty} h(k) \cdot e^{-j\Omega k} = x(n) \cdot H(e^{j\Omega}) \tag{5.43}$$

where $H(e^{j\Omega})$ is defined as the frequency response of the system.

If we compare the definition of the frequency response, as shown in (5.44), and the definition of the transfer function for a system, as shown in (5.45), we see a striking similarity:

$$H(e^{j\Omega}) = \sum_{k=-\infty}^{\infty} h(k) \cdot e^{-j\Omega k} \tag{5.44}$$

$$H(z) = \sum_{k=-\infty}^{\infty} h(k) \cdot z^{-k} \tag{5.45}$$

We can take advantage of this convenient similarity by defining the frequency response of a system in terms of the transfer function by simply allowing z to be replaced by $e^{j\Omega}$. That is,

$$H(e^{j\Omega}) = H(z)\Big|_{z=e^{j\Omega}} \tag{5.46}$$

Example 5.4 Determining the Frequency Response

Problem: Determine the frequency response of the system described in Example 5.3, which has a transfer function of

$$H(z) = \frac{z}{(z - b_1)}$$

Assume that b_1 has a value of 0.8 for this system.

Solution: The frequency response can be found from the transfer function by simply making the substitution of $z = e^{j\Omega}$:

$$H(e^{j\Omega}) = H(z)\big|_{z=e^{j\Omega}} = \frac{e^{j\Omega}}{(e^{j\Omega} - 0.8)} = \frac{\cos(\Omega) + j \cdot \sin(\Omega)}{\cos(\Omega) - 0.8 + j \cdot \sin(\Omega)}$$

$$H(e^{j\Omega}) = \frac{1.0\angle\Omega}{\sqrt{(\cos(\Omega) - 0.8)^2 + (\sin(\Omega))^2}}$$

The values for the frequency response can then be calculated by allowing the frequency Ω to range from 0 to π, as shown in Table 5.5. Note that the complex exponential $e^{j\Omega}$ can be converted to a complex number in the rectangular form of $\cos(\Omega) + j\sin(\Omega)$ using Euler's relationship or converted to polar form as $1 \angle \Omega$.

Table 5.5

Frequency Response

Frequency	*Magnitude*	*Phase (deg)*
0	5.00	0.00
π/4	1.42	–52.5
π/2	0.78	–38.7
3π/4	0.60	–19.9
π	0.56	0.00

Example 5.4 shows that the frequency response of a system is nothing more than a complex valued function of frequency. At any particular frequency, the complex value of the function can be determined and converted to polar form, as shown below:

$$H(e^{j\Omega}) = M(\Omega)\angle\Phi(\Omega) \tag{5.47}$$

In this expression, $M(\Omega)$ represents the adjustment in magnitude that a signal will experience as it passes through the system, and $\Phi(\Omega)$ represents the adjustment in phase that the input signal will experience. For instance, if a digital signal

$$x(n) = 20\cos\left(\frac{\pi \cdot n}{2} + 30°\right) \tag{5.48}$$

was applied to the system of Example 5.4, the output signal could be determined by multiplying the magnitude of the input by the magnitude of the frequency response at $\Omega = \pi/2$, and by shifting the phase of the input by the phase of the frequency response at $\Omega = \pi/2$. The result could be written as

$$y(n) = 20 \cdot 0.78 \cdot \cos\left[(\pi \cdot n/2) + 30° - 38.7°\right] \tag{5.49}$$

or

$$y(n) = 15.62 \cdot \cos\left[(\pi \cdot n/2) - 8.7°\right] \tag{5.50}$$

Using a digital frequency of $\Omega = \pi/2$ may not be comfortable for analog filter designers who are used to working with much larger frequencies. Therefore it is important to note that this digital frequency does relate to some analog frequency through the formula expressed in (5.36). That equation is restated here in terms of the equivalent analog frequency f_d. We see that this frequency is a function of both the digital frequency and the sampling frequency for the system, as discussed earlier:

$$f_d = \frac{\Omega}{2\pi} \cdot f_s \tag{5.51}$$

Therefore, if we are dealing with a system with a sampling frequency of 10,000 samples/second, it may be more natural to think of the input signal above as being an analog signal of 2,500 Hz that has been converted to a digital frequency:

$$f_d = \frac{\pi/2}{2\pi} \cdot 10{,}000 = 2{,}500\ \text{Hz} \tag{5.52}$$

Working with discrete-time systems will require us to be able to use all the material discussed thus far in this chapter. Some problems will be better expressed in the time domain, while others will be easier to work with in the frequency domain. Some system analysis or design will be easier using the system's impulse response, while others will require the use of the system's transfer function. As filter designers, we will definitely need to determine the frequency response of our system, and if we are to implement these filters, a system diagram will be handy. So before we leave this section, let's work an example that uses all the facets of discrete systems discussed so far.

Example 5.5 Complete Discrete-Time System Example

Problem: Consider the difference equation for a discrete-time system shown below. Determine the impulse response, transfer function, and frequency response of the system.

$$\begin{aligned} y(n) = {} & 2.0 \cdot y(n-1) - 1.81 \cdot y(n-2) + 0.68 \cdot y(n-3) \\ & + 1.0 \cdot x(n) + 3.0 \cdot x(n-1) + 3.0 \cdot x(n-2) + 1.0 \cdot x(n-3) \end{aligned}$$

Also, find the location of the poles and zeros for the system as well as draw the system diagram. Finally, determine the output of the system if the input is the analog signal $x(t)$ shown below. Assume that the sampling frequency is 20,000 samples per second.

$$x(t) = 10 + 5\cos(2\pi \cdot 2{,}000 \cdot t - 60°) + 20\sin(2\pi \cdot 8{,}000 \cdot t + 30°)$$

Solution: We begin the solution of the problem by transforming the difference equation to determine the transfer function for the system such that

$$\begin{aligned} Y(z) = {} & 2.0 \cdot z^{-1} \cdot Y(z) - 1.81 \cdot z^{-2} \cdot Y(z) + 0.68 \cdot z^{-3} \cdot Y(z) + 1.0 \cdot X(z) \\ & + 3.0 \cdot z^{-1} \cdot X(z) + 3.0 \cdot z^{-2} \cdot X(z) + 1.0 \cdot z^{-3} \cdot X(z) \end{aligned}$$

or

$$\frac{Y(z)}{X(z)} = \frac{1.0 + 3.0 \cdot z^{-1} + 3.0 \cdot z^{-2} + 1.0 \cdot z^{-3}}{1.0 + 2.0 \cdot z^{-1} - 1.81 \cdot z^{-2} + 0.68 \cdot z^{-3}} = H(z)$$

After we convert $H(z)$ to positive powers of z, we factor the transfer function in order to determine the pole and zero locations. From the expression, we see that there are three zeros at $z = -1.0$ and three poles at $z = 0.8$ and $0.6 \pm j0.7$:

$$H(z) = \frac{(z + 1.0)^3}{(z - 0.8) \cdot (z^2 - 1.2 \cdot z + 0.85)}$$

The frequency response can now be easily determined from the transfer function, as indicated below:

$$H(e^{j\Omega}) = \frac{(e^{j\Omega} + 1.0)^3}{(e^{j\Omega} - 0.8) \cdot (e^{j2\Omega} - 1.2 \cdot e^{j\Omega} + 0.85)}$$

If we are to determine the response of our system to the sampled analog signal, we need to convert the critical analog frequencies to digital frequencies. The DC term is zero frequency in either case, while the analog frequency of 2,000 Hz converts to a digital frequency of π/5, and the analog frequency of 8,000 Hz converts to a digital frequency of 4π/5. The magnitude and phase responses for this $H(e^{j\Omega})$ are shown in Table 5.6.

Table 5.6

Frequency Response

Frequency	*Magnitude*	*Phase (deg)*
0.0	61.54	0
π/5	37.82	–88
2π/5	6.14	–249
3π/5	0.63	–261
4π/5	0.05	–266
π	0.00	–270

By applying the magnitude and phase adjustments indicated in Table 5.6 for frequencies of 0, π/5, and 4π/5, we can determine the output of the system as:

$$y(t) = 615 + 189 \cdot \cos(2\pi \cdot 2{,}000 \cdot t - 148°) + 1.00 \cdot \sin(2\pi \cdot 8{,}000 \cdot t - 236°)$$

The impulse response of the system can be determined by finding the inverse *z*-transform of the transfer function. The most commonly used method for finding the inverse transform is by using partial fraction expansion of *H*(*z*)/*z* and then matching the resulting terms to ones in the transform table.

$$\frac{H(z)}{z} = \frac{(z+1)^3}{z \cdot (z-0.8) \cdot (z^2 - 1.2 \cdot z + 0.85)}$$

After using the standard methods for evaluation of coefficients, we find

$$\frac{H(z)}{z} = \frac{-1.4706}{z} + \frac{13.755}{(z-0.8)} - \frac{11.284 \cdot z - 7.5372}{(z^2 - 1.2 \cdot z + 0.85)}$$

then we write *H*(*z*) as the sum of three terms

$$H(z) = -1.4706 + \frac{13.755 \cdot z}{(z-0.8)} - \frac{11.284 \cdot z^2 - 7.5372 \cdot z}{(z^2 - 1.2 \cdot z + 0.85)}$$

The first two terms can be easily inverse transformed by matching terms in the *z*-transform table. However, the third term relates to a damped sinusoid and requires some manipulation before it can be inverse transformed.

$$\frac{11.284 \cdot z^2 - 7.5372 \cdot z}{(z^2 - 1.2 \cdot z + 0.85)} = \frac{A \cdot \cos(\phi) \cdot z^2 - A \cdot a \cdot \cos(\phi - \Omega) \cdot z}{z^2 - 2 \cdot a \cdot \cos(\Omega) \cdot z + a^2}$$

By comparing denominator terms, we find that $a = 0.92195$ and $\Omega = 0.86217$, while the numerator terms provide $A = 11.337$ and $\phi = 0.09677$. With these values, the transfer function can be inverse transformed to

$$\begin{aligned} h(n) = {} & -1.4706 \cdot \delta(n) + 13.755 \cdot (0.8)^n \cdot u(n) \\ & - 11.337 \cdot (0.92195)^n \cdot \cos(0.86217 \cdot n + 0.09677) \cdot u(n) \end{aligned}$$

There was a good deal of algebra and trigonometry required to find the impulse response. If we need only a few of the first terms of the impulse response, or if we want to verify the correctness of our work, there is a useful method that can be applied. If we return to the original $H(z)$ function and perform long division on the fraction, we will obtain a series of terms as shown:

$$H(z) = \frac{z^3 + 3.0 \cdot z^2 + 3.0 \cdot z + 1.0}{z^3 - 2.0 \cdot z^2 + 1.81 \cdot z - 0.68}$$

$$H(z) = 1.0 + 5.00 \cdot z^{-1} + 11.19 \cdot z^{-2} + 15.01 \cdot z^{-3} + \cdots$$

By inverse transforming this sequence, we have $h(n)$ represented as a series of delayed impulse functions. This will tell us explicitly what the value of the impulse response is for the first few values of the sequence. These values indeed check with those given by the general expression above for $h(n)$:

$$h(n) = \delta(n) + 5.00 \cdot \delta(n-1) + 11.19 \cdot \delta(n-2) + 15.01 \cdot \delta(n-3) + \cdots$$

5.5 PLAYING DIGITIZED WAVEFORMS ON A COMPUTER SYSTEM

In order to get the full benefit of the work we will be doing in the remainder of this text, a computer sound card should be available. Certainly, the C code in this text for the design and implementation of analog and digital filters is the primary incentive for obtaining a copy of this text, but without a sound card to play the sound files that we will process in the next few chapters, an important experience will be missed. We will be using several sound files to illustrate the effects produced by the digital filters which we will be designing. These sound files have been included on the software disc with this text in the \C_CODE\SOUND directory.

In addition to sound file formats, there are other variations that must be considered when playing sound files. First, the sampling frequency must be considered. The standard sampling frequency for studio-quality audio signals is 44,100 Hz. This frequency is high enough to allow audio frequencies in the 20,000 Hz range to be included in the signal information. These frequencies represent the upper limit in the human hearing range. However, not all applications require this level of frequency response. Therefore, sampling rates of 22,050 and 11,025 Hz are also common. These lower frequencies provide attractive alternatives for signals without high frequency components or signals including only speech. The sound cards automatically include an antialiasing filter set to the correct frequency based on the sampling rate. Most sound cards also include the option of selecting the quantization to be used when the signal is sampled. Either 8-bit (1 byte) or 16-bit (2 bytes) resolution can be selected. (Other options are common for industrial applications.) Table 5.7 provides a look at the size of sound files as a function of sampling rate, number of channels, and quantization method.

Table 5.7

Comparison of Sound File Size for 1 Minute of Recording

Quantization	*Channels*	*Sample Rate*	*File Size*
8 bits	Mono (1)	11,025 Hz	0.662 MB
8 bits	Mono (1)	22,050 Hz	1.323 MB
8 bits	Mono (1)	44,100 Hz	2.646 MB
8 bits	Stereo (2)	11,025 Hz	1.313 MB
8 bits	Stereo (2)	22,050 Hz	2.646 MB
8 bits	Stereo (2)	44,100 Hz	5.292 MB
16 bits	Mono (1)	11,025 Hz	1.323 MB
16 bits	Mono (1)	22,050 Hz	2.646 MB
16 bits	Mono (1)	44,100 Hz	5.646 MB
16 bits	Stereo (2)	11,025 Hz	2.646 MB
16 bits	Stereo (2)	22,050 Hz	5.646 MB
16 bits	Stereo (2)	44,100 Hz	10.58 MB

Obviously, the size of audio files can grow very large! That is why it is important to select the sampling rate, number of channels, and quantization method carefully to provide the level of accuracy appropriate to the project. Of course today there are many options for the compression of this data, but there is still a direct correlation between file size and sampling options. There are two different files included with the software disc. The first file, SPEECH, is a monaural file that uses a sampling rate of 11,025 samples per second with 8 bits per sample. The second selection, MUSIC, is also a monaural signal recorded by sampling at 22,050 samples per second with 16 bits per sample. Since most sound cards will also record signals, other test signals can be captured and tested on the computer system as desired.

At this point, it is time to check out the sound card on the computer by playing the sound files mentioned above. It is recommended that the sound files be copied to the hard disk for faster access. We'll be using them as input samples to be designed in the next two chapters.

5.6 CONCLUSION

We have reached the end of our review of discrete-time systems. In this chapter, we found that the complete characteristics of a discrete-time system could be determined by knowing the system's difference equation, impulse response, transfer function, or system diagram. We learned ways to determine any one of these descriptions from any of the others. In addition, we learned how to find the frequency response of a system by direct substitution into the system's transfer function. We will use the material presented in this chapter to design and implement digital filters in the next two chapters. The remainder of this text will be presented in a manner emphasizing application rather than theory, but if the need arises, we can use the material in this chapter to better understand any problems that we might encounter.

Chapter 6

Infinite Impulse Response Digital Filter Design

There are a variety of methods that can be used to design digital filters as we will see in this chapter and the next. One commonly used method is to use the analog filter approximation functions that have already been developed and simply translate them in a way that will make them usable for discrete-time systems. This method, which will be studied in this chapter, makes use of the large backlog of filter design theory and tables of transfer functions that are readily available. Most of the filters designed using this method will be recursive in nature. That is, the output of the filter will depend on previous values of the output (as well as past and current values of the input). These types of filters can theoretically have impulse responses that continue forever and therefore are commonly referred to as infinite impulse response (IIR) filters.

Another method of designing discrete-time filters will be discussed in the next chapter. That method does not depend on analog filter theory, but rather uses the frequency response of the desired filter to directly determine the digital filter coefficients. The method generally yields nonrecursive filters that have outputs depending only on past and current values of the input. These types of filters generally have an impulse response containing only a finite number of values and thus are commonly called finite impulse response (FIR) filters. As we are about to see, both the IIR and FIR design methods will differ from the analog filter design techniques studied in the first part of the text. (A more complete comparison of IIR and FIR filters will be given in Section 8.1.)

In the first three sections of this chapter, we will investigate different methods of translating an analog filter's characteristics into those of a digital filter. As we will see, there is no perfect digital equivalent to an analog filter at all frequencies; however, we can develop filters that closely match the important filter characteristics. In the final section of this chapter, we will develop the C code necessary to evaluate the frequency response characteristics of IIR digital filters.

6.1 IMPULSE RESPONSE INVARIANT DESIGN

The impulse response invariant design method (or impulse invariant transformation) is based on creating a digital filter with an impulse response that is a sampled version of the impulse response of the analog filter. We first start with an analog filter's transfer function $H(s)$, and by using the inverse Laplace transform, we determine the system's continuous impulse response $h(t)$. We next sample that response to determine the system's discrete-time impulse response $h(nT)$. We then take the z-transform of this sampled impulse response to find the discrete-time transfer function $H(z)$. As an illustration, consider the following example.

Example 6.1 Impulse Response Invariant Transformation

Problem: Assume that we wish to convert the following continuous-time transfer function to a discrete-time transfer function using the impulse invariant transformation method:

$$H(s) = \frac{12}{(s+2)\cdot(s+5)}$$

Solution: We first use basic partial fraction expansion techniques to write the transfer function in a form suitable for inverse transformation:

$$H(s) = \frac{4}{(s+2)} - \frac{4}{(s+5)}$$

Then recognizing the Laplace transform pair

$$L^{-1}\left\{H(s) = \frac{A}{(s+a)}\right\} = A\cdot e^{-at}\cdot u(t) = h(t)$$

we can easily find the impulse response as

$$h(t) = (4\cdot e^{-2t} - 4\cdot e^{-5t})\cdot u(t)$$

If we then sample this impulse response at intervals of T, we will have the discrete-time impulse response. Effectively, we simply replace every t with nT to denote the nth sample at intervals of T:

$$h(nT) = (4\cdot e^{-2nT} - 4\cdot e^{-5nT})\cdot u(nT)$$

This expression can be rewritten in a form that more clearly indicates the exponential relationship of *n*:

$$h(nT) = [4 \cdot (e^{-2T})^n - 4 \cdot (e^{-5T})^n] \cdot u(nT)$$

Now we use the *z*-transform table as developed in Chapter 6 to find the transfer function in the *z*-domain:

$$H(z) = \frac{4}{1 - e^{-2T} z^{-1}} - \frac{4}{1 - e^{-5T} z^{-1}}$$

And, finally, we can combine the terms over a common denominator to produce the final result, which can be simplified once a value of the sampling period *T* is chosen:

$$H(z) = \frac{4 \cdot (e^{-2T} - e^{-5T}) \cdot z^{-1}}{(1 - e^{-2T} z^{-1}) \cdot (1 - e^{-5T} z^{-1})}$$

Although Example 6.1 clearly indicates the steps required to translate an analog transfer function to a digital transfer function, we can skip some of the steps by recognizing the common relationship between *H*(*s*) and *H*(*z*). As can be verified in the example, for every term in the analog transfer function of the form shown in (6.1), there is a term created in the digital transfer function of the form shown in (6.2):

$$H(s) = \frac{1}{s + a} \tag{6.1}$$

$$H(z) = \frac{1}{1 - e^{-aT} \cdot z^{-1}} \tag{6.2}$$

This matching technique can also be applied to quadratic terms that have complex roots, where each factor is simply treated individually. For example, if we have a quadratic of the form (6.3), the resulting discrete-time equivalent could then be written and simplified as shown in (6.3) to (6.6):

$$H(s) = \frac{\beta}{(s + \alpha + j\beta) \cdot (s + \alpha - j\beta)} = \frac{j/2}{s + \alpha + j\beta} - \frac{j/2}{s + \alpha - j\beta} \tag{6.3}$$

$$H(z) = \frac{j/2}{1 - e^{-(\alpha + j\beta)T} \cdot z^{-1}} - \frac{j/2}{1 - e^{-(\alpha - j\beta)T} \cdot z^{-1}} \tag{6.4}$$

$$H(z) = \frac{(j/2) \cdot (e^{-(\alpha + j\beta)T} - e^{-(\alpha - j\beta)T}) \cdot z^{-1}}{(1 - e^{-(\alpha + j\beta)T} \cdot z^{-1}) \cdot (1 - e^{-(\alpha - j\beta)T} \cdot z^{-1})} \tag{6.5}$$

$$H(z) = \frac{e^{-\alpha T} \cdot \sin(\beta T) \cdot z^{-1}}{1 - 2 \cdot e^{-\alpha T} \cdot \cos(\beta T) \cdot z^{-1} + e^{-2\alpha T} \cdot z^{-2}} \tag{6.6}$$

Example 6.2 Butterworth Impulse Invariant Filter Design

Problem: Determine the impulse invariant digital filter for a second-order Butterworth approximation function, as shown below. Notice that $H(s)$ is normalized and therefore has a passband edge frequency of 1 rad/sec or (1/2π Hz). Determine the differences that result from choosing sampling periods of T = 1.0 sec and T = 0.1 sec.

$$H(s) = \frac{1}{s^2 + 1.4142 \cdot s + 1}$$

Solution: We can first factor the analog transfer function and use partial fraction expansion to determine

$$H(s) = \frac{j0.7071}{s + 0.7071 + j0.7071} - \frac{j0.7071}{s + 0.7071 - j0.7071}$$

The digital transfer function can then be determined by using the results indicated in (6.3) to (6.6):

$$H(z) = \frac{1.4142 \cdot e^{-0.7071 \cdot T} \cdot \sin(0.7071 \cdot T) \cdot z^{-1}}{1 - 2 \cdot e^{-0.7071 \cdot T} \cdot \cos(0.7071 \cdot T) \cdot z^{-1} + e^{-1.4142 \cdot T} \cdot z^{-2}}$$

We can now make the substitution of the different sampling periods in the general form to find the two distinct transfer functions:

$$H(z)\Big|_{T=1.0} = \frac{.45300 \cdot z^{-1}}{1 - 0.74971 \cdot z^{-1} + 0.24312 \cdot z^{-2}}$$

$$H(z)\Big|_{T=0.1} = \frac{0.093096 \cdot z^{-1}}{1 - 1.85881 \cdot z^{-1} + 0.86812 \cdot z^{-2}}$$

It is interesting to compare the two transfer functions of Example 6.2. We notice first that the gains and pole positions are different solely from the selection of the sampling period (or frequency). We can also get a quick indication of the magnitudes of these transfer functions by determining the response at zero frequency. We can do that easily letting $z = e^{j0} = 1$:

$$H(e^{j0})\Big|_{T=1.0} = \frac{0.45300}{0.49341} = 0.91808$$

$$H(e^{j0})\Big|_{T=0.1} = \frac{9.3096 \cdot 10^{-2}}{9.3174 \cdot 10^{-3}} = 9.9917$$

With this quick check, we see that the response at zero frequency seems to be proportional to 1/*T*. Although using only two values of sampling frequency hardly makes a case, it is true in general that the magnitude is proportional to 1/*T*. For this reason, most impulse invariant designs scale the transfer function by an amount equal to the sampling period. As we can see, if that scaling were used in the previous example, the responses at zero frequency would be very close to unity.

The complete frequency responses for both transformations are shown in Figure 6.1 (with the *T* scaling factor applied). The responses are notably different as we would expect since the two transfer functions have different gains and pole locations. It is important to notice how this variation in transfer function form and frequency response is due solely to the value of sampling period (or sampling frequency) that has been selected.

In order to see why this selection produces the variations, we must remind ourselves of the relationship between the digital and equivalent analog frequencies. As described in Section 6.4, the digital frequency Ω extends from 0 to π where π is analogous to the analog frequency of $f_s/2$ as dictated by the Nyquist criteria. Each point on the frequency axis can be referenced in terms of Ω, which extends from 0 to π, or in terms of f_d, which extends from 0 to $f_s/2$. This relationship can be written in either of the two forms shown in (6.7) and (6.8):

$$\Omega = 2\pi \cdot f_d \,/\, f_s \tag{6.7}$$

$$f_d = f_s \frac{\Omega}{2\pi} \tag{6.8}$$

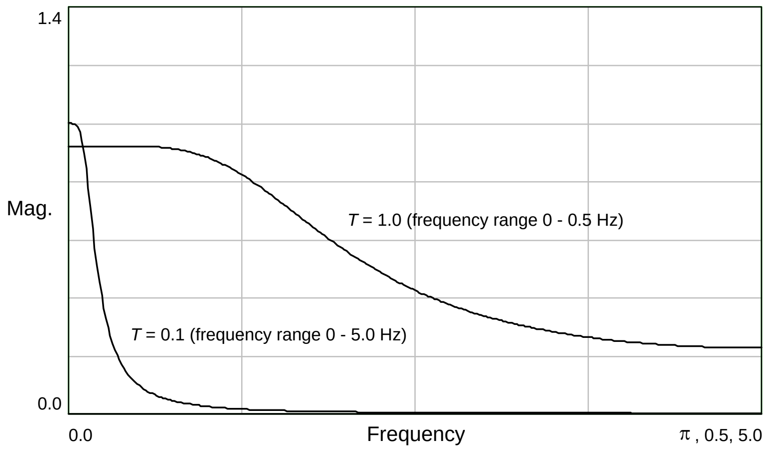


Figure 6.1 Frequency responses for Example 6.2.

Now we are able to see more clearly why the responses are so different. Although the frequency axis extends from 0 to π, it represents different analog frequencies for the two responses. In the case of the $T = 1$ sec response ($f_S = 1$ Hz), the digital frequency range extends from 0 to 0.5 Hz, and the passband edge frequency of 0.159 Hz is clearly visible at a point approximately one-third of the way along the frequency axis. However, in the case of the $T = 0.1$ second response ($f_S = 10$ Hz), the digital frequency range is actually from 0 to 5 Hz. Therefore, the break frequency of 0.159 Hz occurs at a point much closer to the zero frequency point.

Clearly then from this example it is important to pick the sampling frequency for an impulse invariant design carefully. The frequency range of the input signal must be considered as well as the desired overall response. In general, the impulse invariant design method is best for matching low-frequency system responses.

6.2 STEP RESPONSE INVARIANT DESIGN

Another common method of converting an analog transfer function to the digital domain is to match the step response of both systems. The step response invariant design procedure is much the same as the impulse invariant design, except that we must determine the step response of the analog transfer function before it is sampled and *z*-transformed. We can determine the analog system's step response simply by multiplying $H(s)$ by the transform of the step input, which is $1/s$. In (6.9) we defined the system's step response as $G(s)$:

$$G(s) = H(s) \cdot \frac{1}{s} \tag{6.9}$$

Once we have determined *G*(*s*), we can find the time domain response to the step input *g*(*t*) by using the inverse Laplace transform:

$$g(t) = L^{-1}\{G(s)\} \tag{6.10}$$

Then, the discrete-time step response can be determined by sampling the continuous-time version:

$$g(nT) = g(t)\big|_{t=nT} \tag{6.11}$$

Next, the discrete-time system response to the step input can be determined by using the *z*-transform:

$$G(z) = Z\{g(nT)\} = H(z) \cdot \frac{1}{1 - z^{-1}} \tag{6.12}$$

As shown in (6.12), *G*(*z*) is the product of the discrete-time transfer function *H*(*z*) and the *z*-transform of the step input. Therefore, in order to find *H*(*z*), we simply multiply *G*(*z*) by $(1 - z^{-1})$, as shown:

$$H(z) = G(z) \cdot \left(1 - z^{-1}\right) \tag{6.13}$$

As we review this procedure, we can see that the primary steps are the same as for the impulse invariant transformation, except that we have added one step at the beginning and one step at the end. The new initial step requires that we divide the analog transfer function by *s*, and the new final step requires that we multiply the digital transfer function by $(1 - z^{-1})$.

Example 6.3 Step Response Invariant Transformation

Problem: Assume that we wish to convert the continuous-time transfer function of Example 6.1 to a discrete-time transfer function, but this time we want to use the step response invariant transformation method.

$$H(s) = \frac{12}{(s+2)(s+5)}$$

Solution: We need to determine the system's response to a step input by multiplying *H*(*s*) by the Laplace transform of the step input 1/*s*.

$$G(s) = \frac{12}{s \cdot (s+2) \cdot (s+5)} = \frac{1.2}{s} - \frac{2.0}{s+2} + \frac{0.8}{s+5}$$

We can then easily find the time domain response to the step input by finding the inverse Laplace transform of *G*(*s*) as

$$g(t) = (1.2 - 2.0 \cdot e^{-2t} + 0.8 \cdot e^{-5t}) \cdot u(t)$$

If we sample this response at intervals of *T*, we will have the discrete-time response, as shown below:

$$g(nT) = [1.2 - 2.0 \cdot (e^{-2T})^n + 0.8 \cdot (e^{-5T})^n] \cdot u(nT)$$

Then, using the *z*-transform table in Chapter 5, we can find the *z*-transform of *g*(*nT*) as

$$G(z) = \frac{1.2}{1 - z^{-1}} - \frac{2.0}{1 - e^{-2T} z^{-1}} + \frac{0.8}{1 - e^{-5T} z^{-1}}$$

And by combining the terms over a common denominator, we find that

$$G(z) = \frac{(1.2 - 2.0 \cdot e^{-2T} + 0.8 \cdot e^{-5T}) \cdot z^{-1} + (0.8 \cdot e^{-2T} - 2.0 \cdot e^{-5T} + 1.2 \cdot e^{-7T}) \cdot z^{-2}}{(1 - z^{-1}) \cdot (1 - e^{-2T} z^{-1}) \cdot (1 - e^{-5T} z^{-1})}$$

where *G*(*z*) represents the output of the system to a step input. In order to determine the transfer function of the system, we must remove the effects of the step input 1 / (1 – z^{-1}).

$$H(z) = \frac{(1.2 - 2.0 \cdot e^{-2T} + 0.8 \cdot e^{-5T}) \cdot z^{-1} + (0.8 \cdot e^{-2T} - 2.0 \cdot e^{-5T} + 1.2 \cdot e^{-7T}) \cdot z^{-2}}{(1 - e^{-2T} z^{-1}) \cdot (1 - e^{-5T} z^{-1})}$$

Although the result of the previous example looks quite involved, all of the exponential terms will become constants once the sampling period for the system is selected. We can also use the step invariant design method on the Butterworth filter of Example 6.2.

Example 6.4 Butterworth Step Invariant Filter Design

Problem: Determine the step invariant digital filter for a second-order normalized Butterworth approximation function, as shown below. Determine the differences that result for sampling periods of T = 1.0 sec and T = 0.1 sec.

$$H(s)=\frac{1}{s^2+1.4142\cdot s+1}$$

Solution: Again, we first determine the output of the analog system to an input step function and then use partial fraction expansion to determine the individual terms.

$$G(s)=\frac{1}{s}-\frac{0.5+j0.5}{s+0.7071+j0.7071}-\frac{0.5-j0.5}{s+0.7071-j0.7071}$$

The step response can then be determined by finding the inverse Laplace transform of $G(s)$, as indicated below:

$$g(t)=1.0-(0.5+j0.5)\cdot e^{-(0.7071+j0.7071)t}-(0.5-j0.5)\cdot e^{-(0.7071-j0.7071)t}$$

Then, after sampling at intervals of T, the discrete-time step response is determined to be

$$g(nT)=1.0-(0.5+j0.5)\cdot e^{-(0.7071+j0.7071)nT}-(0.5-j0.5)\cdot e^{-(0.7071-j0.7071)nT}$$

The discrete-time response to the step input can then be determined by using the z-transform.

$$G(z)=\frac{1}{1-z^{-1}}-\frac{0.5+j0.5}{1-e^{(0.7071+j0.7071)T}z^{-1}}-\frac{0.5-j0.5}{1-e^{(0.7071-j0.7071)T}z^{-1}}$$

Now, by combining these terms over a common denominator (and performing a considerable amount of complex algebra), we have the system response to a step input:

$$G(z)=\frac{z^{-1}+e^{-0.707T}[(\sin(0.707\cdot T)-\cos(0.707\cdot T)]\cdot z^{-1}}{(1-z^{-1})\cdot[1-2\cdot e^{-0.707\cdot T}\cos(0.707T)\cdot z^{-1}+e^{-1.414\cdot T}z^{-2}]}$$
$$+\frac{e^{-1.414T}z^{-2}-e^{-0.707T}[\sin(0.707\cdot T)+\cos(0.707\cdot T)]\cdot z^{-2}}{(1-z^{-1})\cdot[1-2\cdot e^{-0.707\cdot T}\cos(0.707\cdot T)\cdot z^{-1}+e^{-1.414\cdot T}z^{-2}]}$$

The discrete-time transfer function $H(z)$ can now be determined by removing the $(1 - z^{-1})$ factor relating to the step input, and we can make the substitution of the different sampling periods in the general form to find the two distinct transfer functions:

$$H(z) = \frac{z^{-1} + e^{-0.707 \cdot T}[(\sin(0.707 \cdot T) - \cos(0.707 \cdot T)] \cdot z^{-1}}{[1 - 2 \cdot e^{-0.707 \cdot T} \cos(0.707 \cdot T) \cdot z^{-1} + e^{-1.414 \cdot T} z^{-2}]} + \frac{e^{-1.414 \cdot T} z^{-2} - e^{-0.707 \cdot T}[\sin(0.707 \cdot T) + \cos(0.707 \cdot T)] \cdot z^{-2}}{[1 - 2 \cdot e^{-0.707 \cdot T} \cos(0.707 \cdot T) \cdot z^{-1} + e^{-1.414 \cdot T} z^{-2}]}$$

$$H(z)\Big|_{T=1.0} = \frac{0.94546 \cdot z^{-1} - 0.45205 \cdot z^{-2}}{1 - 0.74971 \cdot z^{-1} + 0.24312 \cdot z^{-2}}$$

$$H(z)\Big|_{T=0.1} = \frac{0.13643 \cdot z^{-1} - 0.12711 \cdot z^{-2}}{1 - 1.8588 \cdot z^{-1} + 0.86812 \cdot z^{-2}}$$

Note that the pole locations are the same as for the impulse invariant design but that the zero locations have changed. We can compare the frequency responses of these two discrete-time filters as shown in Figure 6.2.

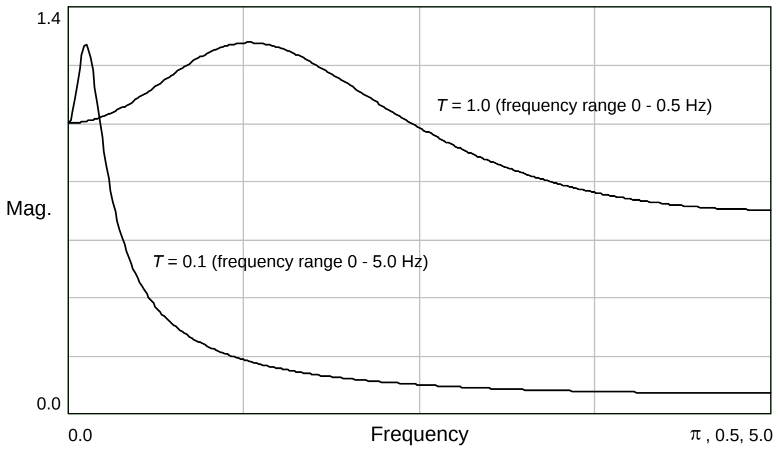


Figure 6.2 Frequency responses for Example 6.4.

As we can see, there is significant difference between the two implementations, but the differences are again a result of the frequency axis having two different scales. In the step invariant design method, there is no need to scale the magnitude as was the case for the impulse invariant design method. By comparing this frequency response to that of Figure 6.1, we see a significant difference. The reason for the difference is the different criteria placed on the design. In the previous section, the emphasis was placed on matching an impulse like input signal, while in this section the aim was to match a step like input signal. As indicated in the figures, the different criteria produce filters with quite different frequency responses. As in the previous section, this method of IIR filter design is best suited to match low-frequency system responses.

6.3 BILINEAR TRANSFORM DESIGN

Both the impulse invariant and step invariant design methods provide good approximations for lowpass and some bandpass analog filter responses. However, they cannot provide good matching of high-frequency responses, which makes it impossible to use them for highpass or bandstop filter design. In fact, they do not provide the best methods for matching analog filter responses when a good match is required throughout a wide range of frequencies. In addition, without careful selection of the sampling frequency and strict band-limiting of the input signal, distortion from aliasing can occur. Therefore, in this section we will discuss the bilinear transformation that endeavors to make a reasonable match over the entire filter frequency range. Of course, that provides a challenge since the analog frequency range extends from zero to infinity and the digital frequency range extends only from zero to π. However, a transformation from the analog *s*-domain to the digital *z*-domain has been developed (as described in more detail in the technical references provided at the end of the text). In this method, the relationship between the *s* and *z* complex variables can be described by the following equation, where *T* is the sampling period:

$$s = \frac{2}{T} \cdot \frac{z-1}{z+1} \tag{6.14}$$

To better understand this relationship, we can represent the complex variable *z* in the exponential form $R \cdot e^{j\Omega}$

$$s = \frac{2}{T} \cdot \frac{R \cdot e^{j\Omega} - 1}{R \cdot e^{j\Omega} + 1} = \frac{2}{T} \cdot \frac{R \cdot \cos\Omega + j \cdot R \cdot \sin\Omega - 1}{R \cdot \cos\Omega + j \cdot R \cdot \sin\Omega + 1} \tag{6.15}$$

This representation can be written in rectangular form as

$$s = \frac{2}{T} \cdot \frac{[(R \cdot \cos\Omega - 1) + j \cdot R \cdot \sin\Omega] \cdot [(R \cdot \cos\Omega + 1) - j \cdot R \cdot \sin\Omega]}{[(R \cdot \cos\Omega + 1) + j \cdot R \cdot \sin\Omega] \cdot [(R \cdot \cos\Omega + 1) - j \cdot R \cdot \sin\Omega]} \tag{6.16}$$

and finally simplified to

$$s = \frac{2}{T} \cdot (\sigma + j\omega) = \frac{2}{T} \cdot \left[\frac{R^2 - 1}{R^2 + 2 \cdot R \cdot \cos\Omega + 1} + j \frac{2 \cdot R \cdot \sin\Omega}{R^2 + 2 \cdot R \cdot \cos\Omega + 1} \right] \tag{6.17}$$

By referring to (6.17) and observing the *s*-plane and *z*-plane in Figure 6.3, we can see that there are three distinct regions in the *s*-domain that relate to three distinct regions in the *z*-domain. In the first case, any point in the *z*-domain that lies outside of the unit circle ($R > 1$) is associated with a point in the right-half plane (RHP) of the *s*-plane ($\sigma > 0$). In the second case, a point in the *z*-domain located inside the unit circle ($R < 1$) is associated with a point in the left-half plane (LHP) of the *s*-domain ($\sigma < 0$). Finally, a point on the unit circle ($R = 1$) is associated with a point in the *s*-plane that lies on the $j\omega$ axis ($\sigma = 0$). In fact, in this last case, the positive $j\omega$ axis relates to the top half of the unit circle as the angle travels from 0 to π, while the negative $j\omega$ axis relates to the bottom half of the unit circle with angles from 0 to $-\pi$.

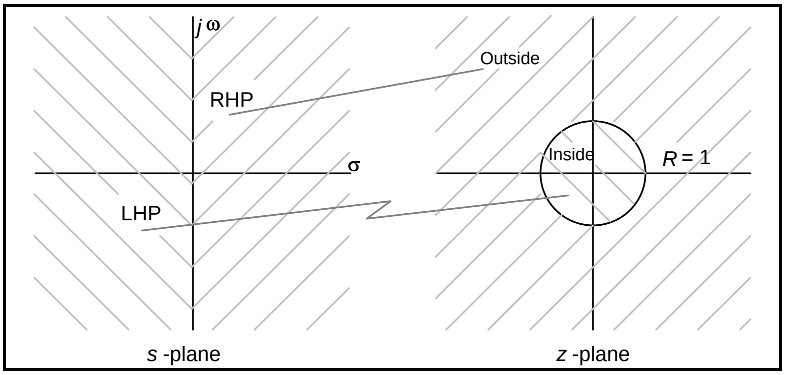


Figure 6.3 Comparison of *s*-plane and *z*-plane using bilinear transform.

Although there does exist a one-to-one relationship between the positive $j\omega$ axis and the upper part of the unit circle, it is a nonlinear one. If we look more closely at the imaginary portion of (6.17) when $R = 1$ (and therefore $\sigma = 0$), we see that

$$\omega = \frac{2}{T} \cdot \frac{\sin(\Omega)}{1 + \cos(\Omega)} = \frac{2}{T} \cdot \tan\left(\frac{\Omega}{2}\right) \tag{6.18}$$

or, in terms of the *z*-domain frequency variable,

$$\Omega = 2 \cdot \tan^{-1}\left(\frac{\omega T}{2}\right) \tag{6.19}$$

Equations (6.18) and (6.19) are important to the bilinear transformation process because they are necessary to determine the proper mapping between the analog and digital domains. This mapping of analog frequencies to digital frequencies is fairly linear for low frequencies, but becomes very nonlinear as higher frequencies are mapped. This mapping of frequencies is often referred to as “warping” to describe how the higher frequencies are warped into their proper place on the unit circle. The reason that this warping is so important is that although we will specify the frequency characteristics of the digital filter by digital frequencies, the filter will be derived from an analog filter transfer function. Therefore, it is necessary to properly determine the analog frequencies to use in the analog design by warping the specified digital frequencies as shown in Example 6.5.

After determining the frequencies necessary for the analog filter design, the filter can be designed using the process described earlier in this text. Once the analog transfer function has been determined, we can use the bilinear transform substitution given in (6.14). Since we will be developing code to implement this transformation process, it is important to carefully describe this substitution. In the case of a first-order factor, the transformation process begins with (6.20).

Example 6.5 Determining Analog and Digital Critical Frequencies

Problem: Assume we wish to design a lowpass digital filter (with sampling frequency of 20 kHz) based upon a Butterworth analog filter. The required characteristics of the *digital* filter are

$$a_{\text{pass}} = -1 \text{ dB},\ a_{\text{stop}} = -20 \text{ dB},\ f_{\text{pass}} = 1 \text{ kHz, and } f_{\text{stop}} = 5 \text{ kHz}$$

What parameters should we use to design the *analog* filter upon which our digital filter will be based?

Solution: We first recognize that the digital frequency axis can be labeled in two different ways, as discussed earlier. Then using (6.7), we can determine

$$\Omega_p = \frac{2 \cdot \pi \cdot 1{,}000}{20{,}000} = 0.1 \cdot \pi$$

$$\Omega_s = \frac{2 \cdot \pi \cdot 5{,}000}{20{,}000} = 0.5 \cdot \pi$$

Once these frequencies have been determined they can be warped using (6.18) to produce the equivalent analog frequencies necessary for the analog filter design.

$$\omega_p = \frac{2}{T} \cdot \tan\left(\frac{\Omega_p}{2}\right) = 2 \cdot \pi \cdot 1{,}008.3 = 6{,}335.4 \quad \text{rad/sec}$$

$$\omega_s = \frac{2}{T} \cdot \tan\left(\frac{\Omega_s}{2}\right) = 2 \cdot \pi \cdot 6{,}366.2 = 40{,}000 \quad \text{rad/sec}$$

Note the slight warping of the lower passband edge frequency (from 1,000 to 1,008 Hz) and the more significant warping of the higher stopband edge frequency (from 5,000 to 6,366 Hz). The attenuations of the filter do not change; therefore, we have enough information to proceed with the design of the analog filter, which will be the subject of the next example.

$$H(z) = \left.\frac{A_1 \cdot s + A_2}{B_1 \cdot s + B_2}\right|_{s=\frac{2}{T}\cdot\frac{z-1}{z+1}} = \frac{A_1 \cdot \left(\frac{2}{T} \cdot \frac{z-1}{z+1}\right) + A_2}{B_1 \cdot \left(\frac{2}{T} \cdot \frac{z-1}{z+1}\right) + B_2} \tag{6.20}$$

In this equation, uppercase *A* and *B* represent the coefficients of the analog filter function. After simplification, (6.21) and (6.22) result with a new set of coefficients where 2/*T* has been replaced with $2 \cdot f_S$. In these equations, lowercase *a* and *b* represent the digital filter coefficients that will be used, and *G* represents the gain adjustment for this first-order term:

$$H(z) = \frac{N_0}{D_0} \cdot \frac{1 + \left(\frac{N_1}{N_0}\right) \cdot z^{-1}}{1 + \left(\frac{D_1}{D_0}\right) \cdot z^{-1}} = G \cdot \frac{a_0 + a_1 \cdot z^{-1}}{b_0 + b_1 \cdot z^{-1}} \tag{6.21}$$

where

$$\begin{aligned} N_0 &= A_2 + 2 f_s A_1 \\ N_1 &= A_2 - 2 f_s A_1 \\ D_0 &= B_2 + 2 f_s B_1 \\ D_1 &= B_2 - 2 f_s B_1 \end{aligned} \tag{6.22}$$

In the case of the quadratic terms that are used to describe our coefficients, the transformation process is shown in (6.23) to (6.25). Again, *G* represents the gain adjustment necessary for each of the quadratic factors for the filter.

$$H(z) = \frac{A_0 \cdot \left(\frac{2}{T}\cdot\frac{z-1}{z+1}\right)^2 + A_1 \cdot \left(\frac{2}{T}\cdot\frac{z-1}{z+1}\right) + A_2}{B_0 \cdot \left(\frac{2}{T}\cdot\frac{z-1}{z+1}\right)^2 + B_1 \cdot \left(\frac{2}{T}\cdot\frac{z-1}{z+1}\right) + B_2} \tag{6.23}$$

$$H(z) = \frac{N_0}{D_0}\cdot\frac{1+\left(\frac{N_1}{N_0}\right)\cdot z^{-1}+\left(\frac{N_2}{N_0}\right)\cdot z^{-2}}{1+\left(\frac{D_1}{D_0}\right)\cdot z^{-1}+\left(\frac{D_2}{D_0}\right)\cdot z^{-2}} = G\cdot\frac{a_0+a_1\cdot z^{-1}+a_2\cdot z^{-2}}{b_0+b_1\cdot z^{-1}+b_2\cdot z^{-2}} \tag{6.24}$$

where

$$\begin{aligned} N_0 &= A_2 + 2f_sA_1 + 4f_s^2A_0 \\ N_1 &= 2\cdot(A_2 - 4f_s^2A_0) \\ N_2 &= A_2 - 2f_sA_1 + 4f_s^2A_0 \\ D_0 &= B_2 + 2f_sB_1 + 4f_s^2B_0 \\ D_1 &= 2\cdot(B_2 - 4f_s^2B_0) \\ D_2 &= B_2 - 2f_sB_1 + 4f_s^2B_0 \end{aligned} \tag{6.25}$$

Example 6.6 Butterworth Bilinear Transform Filter Design

Problem: Determine the digital filter to meet the specifications given in Example 6.5 using the bilinear transformation.

Solution: By entering the attenuations and the prewarped *analog* frequencies in WFilter for analog filter design, we determine the following analog transfer function:

$$H(s) = \frac{7.8877\cdot 10^7}{s^2 + 1.2560\cdot 10^4 s + 7.8877\cdot 10^7}$$

Then, by using the bilinear substitution for *s*, we can determine the transfer function in the digital domain.

$$H(z)=\frac{7.8877\cdot 10^{7}}{\left(\frac{2}{T}\cdot\frac{z-1}{z+1}\right)^{2}+1.2560\cdot 10^{4}\cdot\left(\frac{2}{T}\cdot\frac{z-1}{z+1}\right)+7.8877\cdot 10^{7}}$$

The transfer function can be simplified by using (6.23) to (6.25) to produce

$$H(z)=\frac{0.036161\cdot(1+2\cdot z^{-1}+z^{-2})}{(1-1.3947\cdot z^{-1}+0.53935\cdot z^{-2})}$$

The frequency response of the digital filter designed in Example 6.6 can be determined in the manner discussed in the previous chapter (and is shown in Figure 6.4).

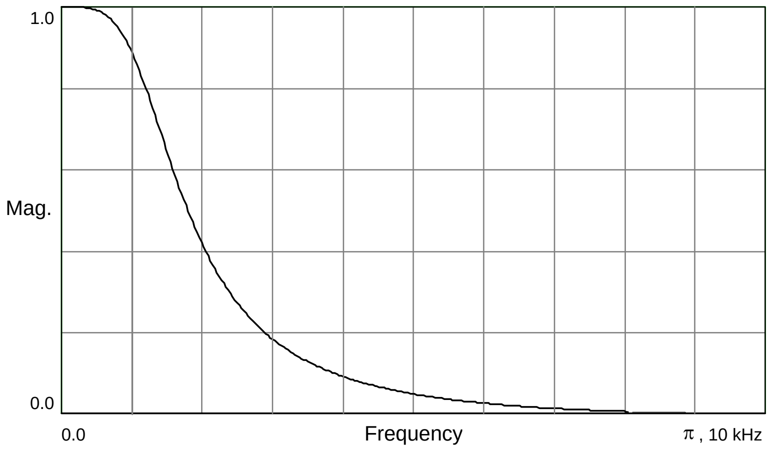


Figure 6.4 Frequency response for Example 6.6.

For a general quadratic factor of the form shown below:

$$H(z)=\frac{a_0+a_1\cdot z^{-1}+a_2\cdot z^{-2}}{b_0+b_1\cdot z^{-1}+b_2\cdot z^{-2}} \tag{6.26}$$

the frequency response can be determined by letting $z = e^{j\Omega}$ as shown:

$$H(e^{j\Omega})=\frac{a_0+a_1\cdot e^{-j\Omega}+a_2\cdot e^{-j2\Omega}}{b_0+b_1\cdot e^{-j\Omega}+b_2\cdot e^{-j2\Omega}} \tag{6.27}$$

The numerator and denominator factors can then be converted.

$$H(e^{j\Omega}) = \frac{[a_0 + a_1 \cos(\Omega) + a_2 \cos(2\Omega)] + j \cdot [a_1 \sin(\Omega) + a_2 \sin(2\Omega)]}{[b_0 + b_1 \cos(\Omega) + b_2 \cos(2\Omega)] + j \cdot [b_1 \sin(\Omega) + b_2 \sin(2\Omega)]} \tag{6.28}$$

We see in Figure 6.4 that the specifications have been met (at 1,000 Hz the –1 dB gain = 0.89125, and at 5,000 Hz the –20 dB gain = 0.1).

Example 6.7 Chebyshev Bilinear Transform Filter Design

Problem: Use WFilter to completely design a Chebyshev digital IIR filter and display the magnitude response. The specifications for this filter are

a_{pass} = –1 dB, a_{stop} = –60 dB,
f_{pass} = 10 kHz, f_{stop} = 20 kHz, and f_{samp} = 50 kHz

Solution: We can supply these values to WFilter and WFilter will calculate the magnitude response of the filter, as shown in Figure 6.5. The digital IIR coefficients and the pole and zero locations are shown in Figure 6.6.

In Figure 6.5 we see the effect of sampling on the frequency response. In particular, we see the lowpass response replicated in mirror image form at the sampling frequency of 50 kHz. In addition, we see the lower half of the reflection at twice the sampling frequency of 100 kHz. If we had chosen a larger-frequency scale, we would see these replications reproduced at all multiples of the sampling frequency. Of course, in the typical discrete-time system, the antialiasing filter will be set to one-half of the sampling frequency (25 kHz in this case), and the user would not see the effects of the higher-frequency components.

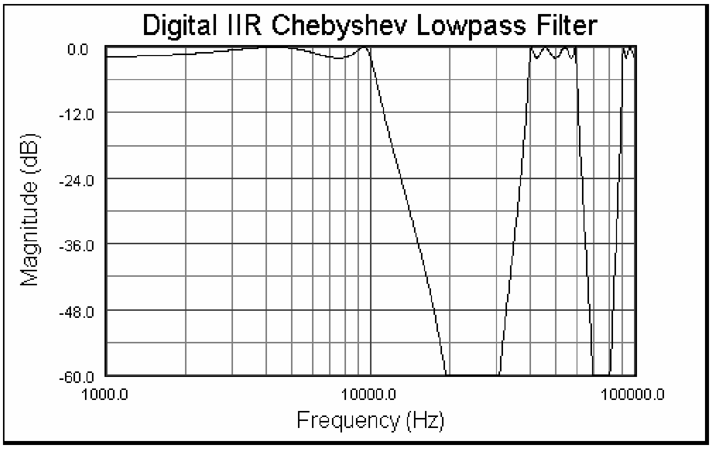


Figure 6.5 Magnitude response from WFilter.

```
Example 6.7 - Chebyshev Lowpass Filter

Selectivity:        Lowpass
Approximation:      Chebyshev
Implementation:     IIR (digital)
Passband gain (dB): -2.0
Stopband gain (dB): -60.0
Passband freq (Hz): 10000.0
Stopband freq (Hz): 20000.0
Sampling freq (Hz): 50000.0

Filter Length/Order: 04
Overall Filter Gain: 1.86714451145E-02

            Numerator Coefficients
QD [ 1 +      z^-1       +       z^-2      ]
== ========================================
01 1.0  2.00000000000E+00  1.00000000000E+00
02 1.0  2.00000000000E+00  1.00000000000E+00

           Denominator Coefficients
QD [ 1 +      z^-1       +       z^-2      ]
== ========================================
01 1.0 -6.20696688131E-01  8.14430976062E-01
02 1.0 -1.18935540161E+00  5.04413209263E-01

                     Zeros
QD [       Real       ] [       Imag       ]
== ========================================
01  -1.00000000000E+00    0.00000000000E+00
02  -1.00000000000E+00    0.00000000000E+00
03  -1.00000000000E+00    0.00000000000E+00
04  -1.00000000000E+00    0.00000000000E+00

                     Poles
QD [       Real       ] [       Imag       ]
== ========================================
01   3.10348344066E-01    8.47416592590E-01
02   3.10348344066E-01   -8.47416592590E-01
03   5.94677700806E-01    3.88293241542E-01
04   5.94677700806E-01   -3.88293241542E-01
```

Figure 6.6 Design values from WFilter.

6.4 C CODE FOR IIR FREQUENCY RESPONSE CALCULATION

To verify that all requirements have been met, we must calculate the frequency response of our filter. In this case, the `Calc_DigIIR_Resp` function shown in Listing 6.1 would be called to perform the frequency response calculations. Computations are made in the same manner as in `Calc_Analog_Resp` except the real and imaginary values are calculated using (6.28).

```
/*======================================================
  Calc_DigIIR_Resp() - calcs response for IIR filters
  Prototype:  int Calc_DigIIR_Resp(Filt_Params *FP,
                                   Resp_Params *RP);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
              RP - ptr to struct holding respon params
======================================================*/
int Calc_DigIIR_Resp(Filt_Params *FP,Resp_Params *RP)
{ int     c,f,q;         /*  loop counters */
  double  rad2deg,       /*  rad to deg conversion */
          omega,omega2,  /*  radian freq and square  */
          rea,img;       /*  real and imag part */

  rad2deg = 180.0 / PI; /*  set rad2deg */
  /*  Loop through each of the frequencies */
  for(f = 0 ;f < RP->tot_pts; f++)
  { /*  Initialize magna and angle */
    RP->magna[f] = FP->gain;
    RP->angle[f] = 0.0;
    /*  Pre calc omega and omega squared */
    omega = PI2 * RP->freq[f] / FP->fsamp;
    omega2 = 2 * omega;
    /*  Loop through coefs for each quadratic */
    for(q = 0 ;q < (FP->order+1)/2; q++)
    { /*  c is coef index = 3 * quad index */
      c = q * 3;
      /* Numerator values */
      rea = FP->acoefs[c] + FP->acoefs[c+1]*cos(omega)
                        + FP->acoefs[c+2]*cos(omega2);
      img = -FP->acoefs[c+1]*sin(omega)
                        - FP->acoefs[c+2]*sin(omega2);
      RP->magna[f] *= sqrt(rea*rea + img*img);
      RP->angle[f] += atan2(img,rea);
      /* Denominator values */
      rea = FP->bcoefs[c] + FP->bcoefs[c+1]*cos(omega)
                        + FP->bcoefs[c+2]*cos(omega2);
      img = -FP->bcoefs[c+1]*sin(omega)
                        - FP->bcoefs[c+2]*sin(omega2);
      RP->magna[f] /= sqrt(rea*rea + img*img);
      RP->angle[f] -= atan2(img,rea);
    }
    /* Convert to degrees */
    RP->angle[f] *= rad2deg;
  }
  /*  Convert magnitude response to dB if indicated */
  if(RP->mag_axis == LOG)
  { for(f = 0 ;f < RP->tot_pts; f++)
    { /* Handle very small numbers */
      if(RP->magna[f] < ZERO)
      { RP->magna[f] = ZERO;}
      RP->magna[f] = 20 * log10(RP->magna[f]);
    }
  }
  return ERR_NONE;
}
```

Listing 6.1 `Calc_DigIIR_Resp` function.

6.5 CONCLUSION

In this chapter we investigated three different methods of generating digital IIR filters from analog transfer functions. In each case we found that there is no perfect match to the original analog function, primarily because there is a strict limit on the frequency range of a digital filter. However, the bilinear transform method does provide good overall response characteristics, and for that reason it was chosen to be implemented in C code. (Details of this code are provided in Appendix G.) The frequency response characteristics of the IIR filter were also considered, and the calculation of the response was implemented in C code. Finally, we used WFilter to design an IIR filter and display the complete magnitude response including the multiple replications of the original response. The implementation of IIR filters is discussed in Chapter 8.

Chapter 7

Finite Impulse Response Digital Filter Design

In the last chapter, we considered the design of digital filters based on the approximation methods for analog filters. We investigated a number of ways that the transfer functions in the analog domain could be converted to transfer functions in the digital domain. In this chapter, we will develop methods that deal with the digital filter as a unique filter type, not based on analog filter approximation methods. The focus of this chapter will be on finite impulse response (FIR) filters that have only a finite number of terms in their impulse response. These filters have a number of advantages over the IIR filter types. An FIR filter is always stable, realizable, and provides a linear phase response under specific conditions. These characteristics make FIR filters attractive to many filter designers. However, the major disadvantage of FIR filters is that the number of coefficients needed to implement a specific filter is often much larger than for IIR designs. A more complete comparison of IIR and FIR filters is given in Section 8.1.

We will begin this chapter with a standard method of designing FIR digital filters using the Fourier series description of the desired frequency response. This method will then be modified and improved by using a windowing technique to improve the shape of the responses. In addition, the Parks-McClellan optimization technique will be discussed as a technique of reducing the length of the resultant FIR filters. Finally, the C code for determining the frequency response of FIR filters will be developed.

7.1 USING FOURIER SERIES IN FILTER DESIGN

There are a number of methods that could be used to design FIR filters. We will investigate one of the most popular in this section. Other methods are described in the references listed in Appendix A for digital filter design.

7.1.1 Frequency Response and Impulse Response Coefficients

In the process of filter design, the designer begins with the frequency response characteristics. The critical band edge frequencies and the gains within each band are determined to meet certain specifications. We have found that the frequency response for digital filters is actually periodic in the frequency domain with a period of the sampling frequency. For example, a typical lowpass filter specification is shown in Figure 7.1, which clearly indicates the periodic nature of the frequency response. Since this response is periodic, it can be described by a Fourier series of the form shown in (7.1). In this formulation, the complex frequency exponential is allowed to take on all possible frequency values.

$$H(e^{j\Omega}) = \sum_{k=-\infty}^{\infty} h(k)\cdot e^{-jk\Omega} \tag{7.1}$$

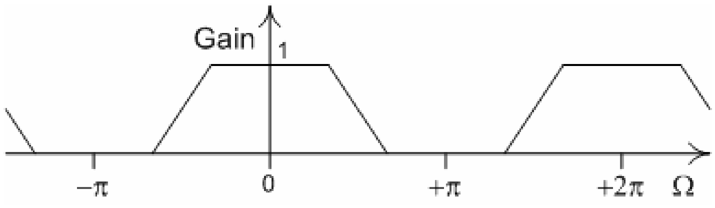


Figure 7.1 Periodic digital frequency response.

The coefficients within the summation are the impulse response coefficients that describe the digital FIR filter. The procedure for determining the impulse response coefficients from the frequency response is straightforward and provided in the digital filter design reference texts listed in Appendix A. The final result of the derivation is shown in (7.2). As indicated by the limits of the integral, the integration must include only one full period of the frequency response.

$$h(n) = \frac{1}{2\pi}\int_{\Omega_o-\pi}^{\Omega_o+\pi} H(e^{j\Omega})\cdot e^{jn\Omega}\cdot d\Omega, \quad n = 0, \pm 1, \pm 2, \ldots \tag{7.2}$$

We will not be able to implement an infinite number of coefficients as (7.2) indicates. The number of coefficients we retain is a compromise between how well we want our design to approximate the ideal, and how many coefficients can be retained because of time delay, implementation cost, or other constraints. We can assume that the indices are limited to the range $-M \le n \le +M$, which limits the number of coefficients retained to $N = 2\,M + 1$. By making this selection, we are in effect setting all other coefficients to zero. Figure 7.2 shows the effect of limiting the number of coefficients by graphing the frequency response using a finite number of coefficients. The frequency response can be determined by using a modified form of (7.1), as shown in (7.3):

$$H(e^{j\Omega}) = \sum_{n=-M}^{M} h(n) \cdot e^{-jn\Omega} \tag{7.3}$$

As we increase the number of coefficients in the FIR filter approximation, we can see that a ripple concentrates near the passband edge frequency. This ripple cannot be eliminated, even by increasing the number of impulse response coefficients; it simply concentrates at the transition. This effect is known as Gibbs's phenomenon and results whenever a discontinuity is modeled with a series. However, as we will see in the next section, there are methods we can use to reduce this effect.

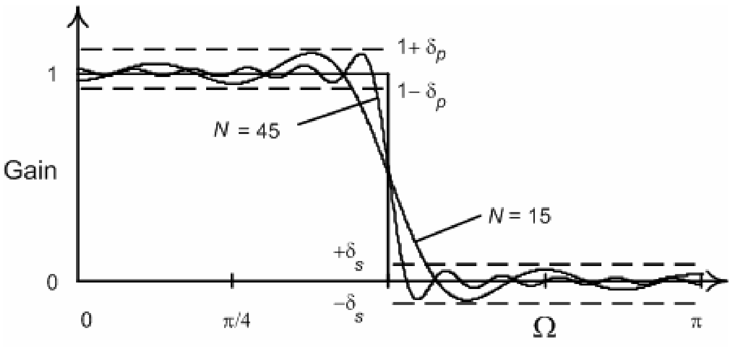


Figure 7.2 Approximated responses to lowpass filter.

Figure 7.2 also shows a more common method of specifying passband and stopband gain for FIR filters. The errors within the passband and stopband are specified as δ_p and δ_s, respectively. As we can see, the frequency response is allowed to fluctuate both positively and negatively within these error limits. We can translate these specifications into the decibel gain specifications with which we are familiar by using (7.4) and (7.5). Alternatively, we can convert our decibel gains into these error values using (7.6) and (7.7).

$$a_{\text{pass}} = 20\log(1-\delta_p) \tag{7.4}$$

$$a_{\text{stop}} = 20\log(\delta_s) \tag{7.5}$$

$$\delta_p = 1 - 10^{0.05 \cdot a_{\text{pass}}} \tag{7.6}$$

$$\delta_s = 10^{0.05 \cdot a_{\text{stop}}} \tag{7.7}$$

As indicated earlier, the phase response of an FIR filter can be a linear function of frequency under certain conditions. For example, if we assume that frequency response within the passband of an FIR filter is as shown in (7.8), we are specifying that the gain must be unity, while the phase angle changes linearly with frequency:

$$H_{\text{passband}}(e^{j\Omega}) = 1 \cdot e^{-j\tau\Omega} = 1\angle - \tau\Omega \tag{7.8}$$

The necessary conditions that allow for this linear phase shift (or constant group delay) are that the impulse response coefficients be either symmetric or antisymmetric and that τ take on the value in (7.9) where *N* is the number of coefficients or the length of the filter. The filter coefficients are symmetric if they satisfy (7.10) and antisymmetric if they satisfy (7.11).

$$\tau = \frac{N-1}{2} \tag{7.9}$$

$$h(n) = h(-n) \tag{7.10}$$

$$h(n) = -h(-n) \tag{7.11}$$

Perhaps it is time to say a few words about the difference between filter length and filter order. Analog and digital IIR filters use the order of the filter as a measure of the filter's "size." The order refers to the highest-order term in the polynomial equation used to describe the filter. On the other hand, digital FIR filters typically use the number of impulse response coefficients required to describe it as its "size." This is probably because most FIR filters are implemented using convolution where the number of coefficients directly affects the length of the processing. Once the FIR coefficients are substituted into a difference equation (a little later in this section), we will see that the length of an FIR filter will simply be one larger than its order.

7.1.2 Characteristics of FIR Filters

When we consider symmetric and antisymmetric coefficients combined with even and odd filter lengths, four different types of FIR filters can be designed. Each of the four types has unique characteristics that can be described briefly as follows.

Type 1 FIR filters. The type 1 FIR filters, which have symmetric coefficients and odd length, also have a frequency response that has even symmetry about both $\Omega = 0$ and $\Omega = \pi$. This even symmetry allows the frequency response to take on any value at these two critical frequencies, and thus lowpass, highpass, bandpass, and bandstop filters can be implemented using this FIR type.

Type 2 FIR filters. The type 2 FIR filters, which have symmetric coefficients and even length, have a frequency response that is even about $\Omega = 0$ and odd about $\Omega = \pi$. This condition dictates that the response at $\Omega = \pi$ be zero and thus type 2 FIR filters are not recommended for highpass or bandstop filters.

Type 3 FIR filters. The type 3 FIR filters, which have antisymmetric coefficients and odd length, have a frequency response that has odd symmetry at both $\Omega = 0$ and $\Omega = \pi$. Because of the odd symmetry, the frequency response of this filter type must be zero at both of these two critical frequencies. Thus, this filter type is not recommended for lowpass, highpass, or bandstop filters. However, this type of filter does provide a 90° phase shift of the output signal with respect to the input and therefore can be used to implement a differentiator or Hilbert transformer. This type of filter has other characteristics that make it the best choice for Hilbert transformation, while the differentiator is usually implemented using a type 4 filter.

Type 4 FIR filters. The type 4 FIR filters, which have antisymmetric coefficients and even length, have a frequency response that has odd symmetry about $\Omega = 0$ and even symmetry about $\Omega = \pi$. The odd symmetry condition makes this type of filter a poor choice to implement either lowpass or bandstop filters. But, just as in the type 3 case, this filter provides a 90° phase shift that makes it able to implement differentiators and Hilbert transformers. This type of filter has better characteristics (in most cases) for implementing a differentiator than type 3, but the type 3 filter has some advantages over this filter type for implementing the Hilbert transform.

Since the type 1 FIR filter can be used to implement any of the filters we need to design, we will discuss only that type of filter from this point forward in this text. Further information concerning the other filter types can be found in a number of the digital filter design references listed in Appendix A.

The filter coefficients derived from (7.2) will not produce a causal filter. This means that the system could not be implemented in real time. We can verify this if we consider the output of a discrete-time system produced by the convolution of the input signal with the impulse response coefficients as shown in (7.12). Notice that the output $y(n)$ becomes only a function of the input $x(n)$ and does not include any past values of the output as in the IIR filter case.

$$y(n) = \sum_{k=-M}^{M} h(k) \cdot x(n-k) \tag{7.12}$$

As we see, using the impulse response coefficients directly will result in $y(n)$ being determined by future values of the input. For example, when $k = -M$, the summation includes a term $x(n + M)$ that refers to an input value M sampling periods ahead of $y(n)$'s reference time. The problem can be handled by shifting all coefficient values to the right on the time axis so that only positive values of n produce coefficients, as shown in Figure 7.3. The disadvantage of this action is to increase the time delay between system input and output by M sampling periods.

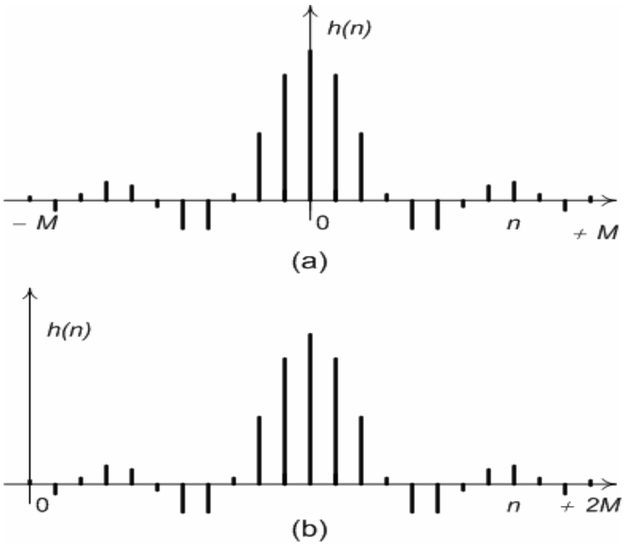


Figure 7.3 (a) Noncausal and (b) causal coefficients.

The causal coefficients can be determined from the noncausal coefficients by making the following index adjustments. As the noncausal coefficients indices take on values from $-M$ to $+M$, the causal coefficient indices will take on values from 0 to 2 M, as shown in (7.13):

$$h_{\text{causal}}(n+M) = h_{\text{noncausal}}(n)\,,\ n = 0, \pm 1, \ldots, \pm M \tag{7.13}$$

7.1.3 Ideal FIR Impulse Response Coefficients

We can now determine the ideal coefficients for various filter types by using the integral formula of (7.2). In each case, figures depicting ideal lowpass, highpass, bandpass, and bandstop filters along with equations for determining the coefficients based on the parameters of the particular filter are given. The frequency response in the passband of each filter is as defined in (7.8) and allows us to determine causal coefficients directly. We will be assuming that the desired passband magnitude response is 1, while the stopband response is 0. We will be using δ_p and δ_s (or a_{pass} and a_{stop}) later to help determine the required length of the filter.

In the first case, Figure 7.4 illustrates the lowpass filter specification with the resulting derivation of the lowpass filter coefficients shown in (7.14) and (7.15). The highpass, bandpass, and bandstop filter cases are portrayed respectively in Figures 7.5 to 7.7 with the appropriate derivations in (7.16) to (7.21). In each case, $\tau = M$ as determined from (7.9).

Example 7.1 Determining Ideal Coefficients for an FIR Filter

Problem: Determine the ideal impulse response coefficients for a lowpass filter of length 21 to satisfy the following specifications:

$$\omega_{\text{pass}} = 2\pi\cdot 3{,}000 \text{ rad/sec}, \ \omega_{\text{stop}} = 2\pi\cdot 4{,}000 \text{ rad/sec, and } f_s = 20 \text{ kHz}$$

Solution: We first need to determine Ω_c, the cutoff frequency for the ideal filter. This frequency can be set in the middle of the transition band and converted to a digital frequency:

$$\Omega_c = (\omega_{\text{stop}} + \omega_{\text{pass}})/(2\cdot f_s) = 1.0996 \quad \text{rad/sec}$$

Using this value along with $\tau = 10$ in (7.15), we can determine the following ideal causal coefficients:

$$\begin{aligned}
h(10) &= 0.35000\\
h(9) = h(11) &= 0.28362\\
h(8) = h(12) &= 0.12876\\
h(7) = h(13) &= -0.01660\\
h(6) = h(14) &= -0.07568\\
h(5) = h(15) &= -0.04502\\
h(4) = h(16) &= 0.01639\\
h(3) = h(17) &= 0.04491\\
h(2) = h(18) &= 0.02339\\
h(1) = h(19) &= -0.01606\\
h(0) = h(20) &= -0.03183
\end{aligned}$$

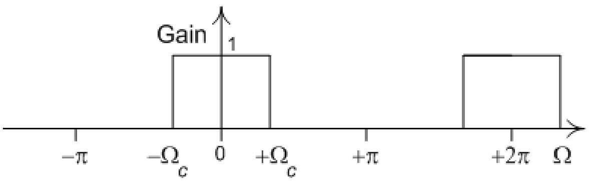


Figure 7.4 Lowpass filter specification.

$$h_{LP}(n) = \frac{1}{2\pi}\int_{-\Omega_c}^{+\Omega_c} e^{j(n-\tau)\Omega}\cdot d\Omega, \qquad (7.14)$$
$$n = 0, 1, \ldots, 2M$$

$$h_{LP}(n) = \begin{cases} \dfrac{\sin[(n-\tau)\Omega_c]}{(n-\tau)\pi}, & \text{for } n \neq \tau \\ \Omega_c/\pi, & \text{for } n = \tau \end{cases}$$
$$n = 0, 1, \ldots, 2M \qquad (7.15)$$

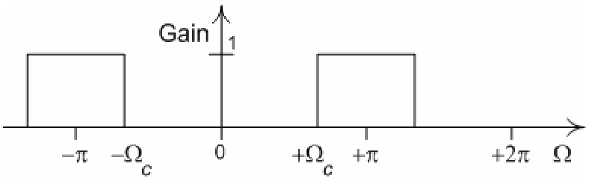


Figure 7.5 Highpass filter specification.

$$h_{HP}(n) = \frac{1}{2\pi}\left[\int_{-\pi}^{-\Omega_c} e^{j(n-\tau)\Omega}\cdot d\Omega + \int_{\Omega_c}^{+\pi} e^{j(n-\tau)\Omega}\cdot d\Omega\right]$$
$$n = 0, 1, \ldots, 2M \qquad (7.16)$$

$$h_{HP}(n) = \begin{cases} \dfrac{\sin[(n-\tau)\pi] - \sin[(n-\tau)\Omega_c]}{(n-\tau)\pi}, & \text{for } n \neq \tau \\ (\pi - \Omega_c)/\pi, & \text{for } n = \tau \end{cases}$$
$$n = 0, 1, \ldots, 2M \qquad (7.17)$$

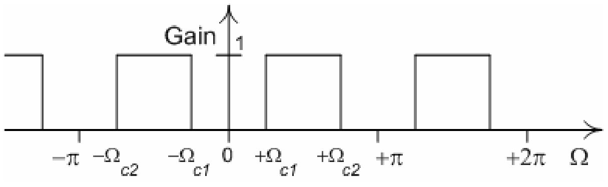


Figure 7.6 Bandpass filter specification.

$$h_{BP}(n) = \frac{1}{2\pi}\left[\int_{-\Omega_{c2}}^{-\Omega_{c1}} e^{j(n-\tau)\Omega}\cdot d\Omega + \int_{+\Omega_{c1}}^{+\Omega_{c2}} e^{j(n-\tau)\Omega}\cdot d\Omega\right]$$
$$n = 0, 1, \ldots, 2M \tag{7.18}$$

$$h_{BP}(n) = \begin{cases} \dfrac{\sin[(n-\tau)\Omega_{c2}] - \sin[(n-\tau)\Omega_{c1}]}{(n-\tau)\pi}, & \text{for } n \neq \tau \\ (\Omega_{c2} - \Omega_{c1})/\pi, & \text{for } n = \tau \end{cases}$$
$$n = 0, 1, \ldots, 2M \tag{7.19}$$

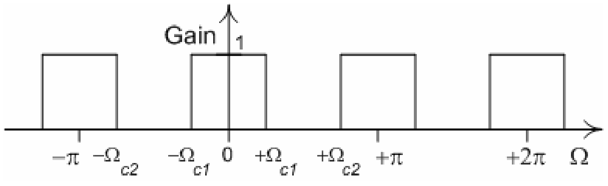


Figure 7.7 Bandstop filter specification.

$$h_{BS}(n) = \frac{1}{2\pi}\left[\int_{-\pi}^{-\Omega_{c2}} e^{j(n-\tau)\Omega}\cdot d\Omega + \int_{-\Omega_{c1}}^{+\Omega_{c1}} e^{j(n-\tau)\Omega}\cdot d\Omega + \int_{+\Omega_{c2}}^{+\pi} e^{j(n-\tau)\Omega}\cdot d\Omega\right]$$
$$n = 0, 1, \ldots, 2M \tag{7.20}$$

$$h_{BS}(n) = \begin{cases} \dfrac{\sin[(n-\tau)\pi] - \sin[(n-\tau)\Omega_{c2}] + \sin[(n-\tau)\Omega_{c1}]}{(n-\tau)\pi}, & \text{for } n \neq \tau \\ (\pi - \Omega_{c2} + \Omega_{c1})/\pi, & \text{for } n = \tau \end{cases}$$
$$n = 0, 1, \ldots, 2M \tag{7.21}$$

7.2 WINDOWING TECHNIQUES TO IMPROVE DESIGN

As indicated in the previous section, we are not able to include the infinite number of coefficients necessary to implement an ideal filter. We will have to reduce the number of coefficients used based on the constraints of our design. In the previous section, we simply truncated all noncausal coefficients beyond the indices $\pm M$ and kept the rest. (We will use the noncausal description of the filter coefficients for mathematical simplicity at this point. After the windowing process has been completed, we can shift the resulting coefficients to produce a causal filter.) This procedure can be compared to placing a window of width $N = 2M + 1$ over all of the ideal coefficients, as shown in Figure 7.8. All of the coefficients within the window are retained and all coefficients outside of the window are discarded. In effect we have produced a rectangular "window" function in which all window coefficients with indices within the range of the window have a value of 1 and all other coefficients have a value of 0. The retained values of the filter coefficients would then be determined by performing a coefficient-by-coefficient multiplication of the ideal coefficients and the window coefficients, as indicated in (7.22):

$$h(n) = h_{\text{ideal}}(n) \cdot w(n),\ n = 0, \pm 1, \ldots, \pm M \tag{7.22}$$

The rectangular window coefficients can be formally defined in (7.23). However, the abrupt truncation of the filter coefficients has an adverse effect on the resulting filter's frequency response. Therefore, a number of other window functions have been proposed which smoothly reduce the coefficients to zero. For example, a simple triangular window (also called the Bartlett window) as shown in Figure 7.9 would smooth the truncation process. An expression for these window coefficients is given in (7.24), where $M = (N - 1)/2$.

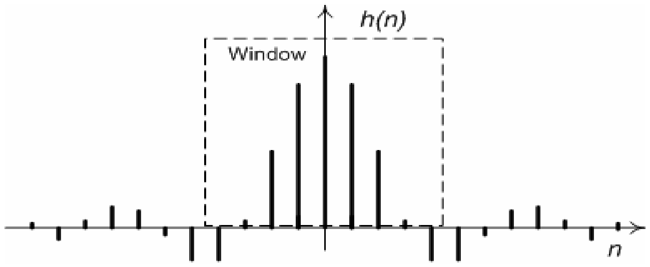


Figure 7.8 Window selection of coefficients.

$$w_{\text{rect}}(n) = w_{\text{rect}}(-n) = 1,\ n = 0, 1, \ldots, M \tag{7.23}$$

$$w_{\text{bart}}(n) = w_{\text{bart}}(-n) = \frac{(M-n)}{M}, n = 0,1,\ldots,M \tag{7.24}$$

Many window functions have been based on the raised cosine function including the von Hann, Hamming, and Blackman windows. Graphs of those windows are also shown in Figure 7.9 with the method of coefficient calculation for each window function provided in (7.25) to (7.27).

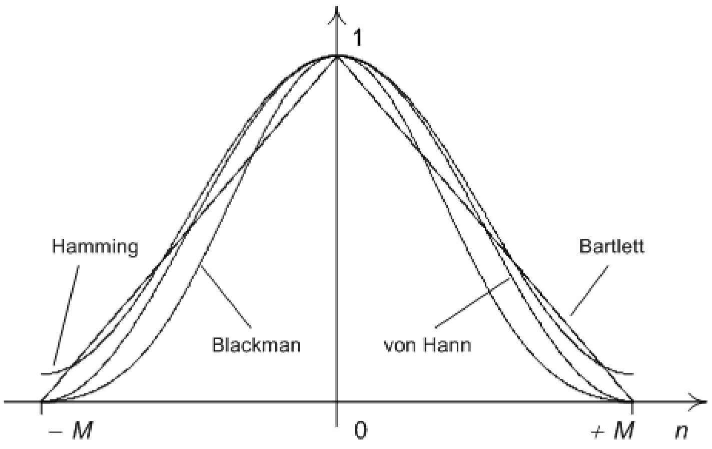


Figure 7.9 Bartlett, Blackman, Hamming, and von Hann windows.

$$w_{\text{hann}}(n) = w_{\text{hann}}(-n) = 0.5\left\{1 - \cos\left[\frac{\pi \cdot (M-n)}{M}\right]\right\},$$
$$n = 0,1,\ldots,M \tag{7.25}$$

$$w_{\text{hamm}}(n) = w_{\text{hamm}}(-n) = 0.54 - 0.46 \cdot \cos\left[\frac{\pi \cdot (M-n)}{M}\right],$$
$$n = 0,1,\ldots,M \tag{7.26}$$

$$w_{\text{blck}}(n) = 0.42 - 0.5 \cdot \cos\left[\frac{\pi \cdot (M-n)}{M}\right] + 0.08 \cdot \cos\left[\frac{2\pi \cdot (M-n)}{M}\right]$$
$$\text{and } w_{\text{blck}}(n) = w_{\text{blck}}(-n), n = 0,1,\ldots,M \tag{7.27}$$

As more time was spent trying to improve the window functions used in FIR filter design, it became apparent for a fixed length of filter that there was a tradeoff between transition band roll-off and attenuation in the stopband. One of the window functions that developed because of this fact was the Kaiser window function, as shown in Figure 7.10. The expression for the window coefficients as given in (7.28) is based on the modified Bessel function of the first kind I_o. The value β generally ranges from 3 to 9 and can be used to control the trade-off between the transition band and stopband characteristics.

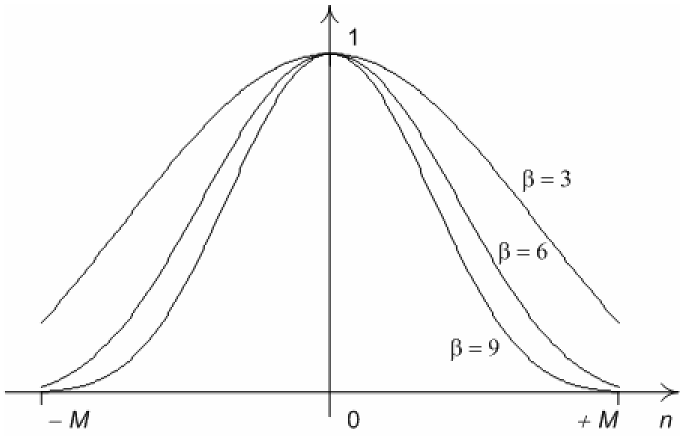


Figure 7.10 Kaiser windows with various β values.

$$w_{\text{kais}}(n) = w_{\text{kais}}(-n) = \frac{I_o\left[\beta \cdot \sqrt{1-\left(\frac{2 \cdot n}{M}\right)^2}\right]}{I_o(\beta)}, \quad n = 0, 1, \ldots, M \tag{7.28}$$

A reasonable estimate of β in the equation above has been determined empirically by Kaiser, as shown in (7.29):

$$\beta = \begin{cases} 0.1102 \cdot (A - 8.7), & \text{for } A > 50 \\ 0.5842 \cdot (A - 21)^{0.4} + 0.07886 \cdot (A - 21), & \text{for } 21 \le A \le 50 \\ 0.0, & \text{for } A < 21 \end{cases} \tag{7.29}$$

In (7.29), the variable A represents the larger of the band errors (δ_p or δ_s) expressed as attenuation, as shown in (7.30):

$$A = -20 \cdot \log[\min(\delta_p, \delta_s)] \tag{7.30}$$

In addition, Kaiser developed empirical estimates of the filter length required to satisfy a given set of filter specifications, as indicated in (7.31). It should be emphasized here that FIR filter design is not as precise as IIR design. The truncation/modification of coefficients results in responses that may or may not meet the requirements. Therefore, the *N* value of (7.31) is just an estimate and the filter responses must be checked carefully to determine if all requirements are met. If they are not, the value on *N* should be adjusted (usually up, but sometimes decreasing *N* can result in a better filter).

$$N = \begin{cases} \dfrac{A - 7.95}{2.285 \cdot \Delta\Omega}, & \text{for } A > 21 \\ \dfrac{5.794}{\Delta\Omega}, & \text{for } A < 21 \end{cases} \tag{7.31}$$

In (7.31), ΔΩ represents the normalized radian transition band for lowpass and highpass filters and the smaller of the two normalized transition bands in the case of bandpass and bandstop filters.

$$\Delta\Omega = \left|\omega_{\text{stop}} - \omega_{\text{pass}}\right| / f_s \tag{7.32}$$

Once the desired window function has been selected and the adjustments made to the ideal coefficients, the causal coefficients can be determined as indicated in the previous section.

Example 7.2 Determining Hamming Coefficients for an FIR Filter

Problem: Determine the coefficients for a lowpass filter using a Hamming window of length 21 to satisfy the specifications shown below:

$\omega_{\text{pass}} = 2\pi \cdot 3{,}000$ rad/sec, $\omega_{\text{stop}} = 2\pi \cdot 4{,}000$ rad/sec, and $f_s = 20$ kHz

Solution: The ideal coefficients have been determined in Example 7.1. We can use (7.24) to determine the noncausal Hamming window coefficients as shown. After multiplication and shifting the coefficients by 10 sampling periods, the causal windowed coefficients result.

Figure 7.11 shows the frequency response for both Examples 7.1 and 7.2. The rectangular window produces a filter that emphasizes transition band roll-off over ripple in the stopband. On the other hand, the filter produced by using Hamming coefficients has no noticeable ripple, but does not have a rapid roll-off in the transition band.

$w(0) = 1.0000$	$h(10) = 0.35000$
$w(1) = w(-1) = 0.97749$	$h(9) = h(11) = 0.27723$
$w(2) = w(-2) = 0.91215$	$h(8) = h(12) = 0.11745$
$w(3) = w(-3) = 0.81038$	$h(7) = h(13) = -0.01345$
$w(4) = w(-4) = 0.68215$	$h(6) = h(14) = -0.05163$
$w(5) = w(-5) = 0.54000$	$h(5) = h(15) = -0.02431$
$w(6) = w(-6) = 0.39785$	$h(4) = h(16) = 0.00652$
$w(7) = w(-7) = 0.26962$	$h(3) = h(17) = 0.01211$
$w(8) = w(-8) = 0.16785$	$h(2) = h(18) = 0.00393$
$w(9) = w(-9) = 0.10251$	$h(1) = h(19) = -0.00165$
$w(10) = w(-10) = 0.0800$	$h(0) = h(20) = -0.00255$

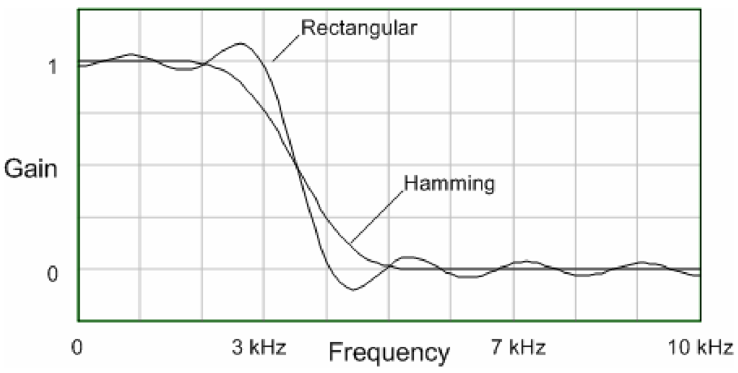


Figure 7.11 Magnitude responses for Examples 7.1 and 7.2.

Example 7.3 Determining Kaiser Coefficients for an FIR Filter

Problem: Determine the impulse response coefficients for a bandpass filter using a Kaiser window to satisfy the following specifications:

f_{pass1} = 4 kHz, f_{pass2} = 5 kHz, f_{stop1} = 2 kHz, f_{stop2} = 8 kHz,
a_{pass1} = –0.5 dB, $a_{\text{stop1}} = a_{\text{stop2}}$ = –50 dB, and f_{samp} = 20 kHz

Solution: The solution to this problem begins with the determination of the passband and stopband errors δ_p and δ_s by using (7.6) and (7.7).

$$\delta_p = 1 - 10^{-0.025} = 0.055939$$

$$\delta_s = 10^{-2.5} = 0.0031623$$

We can then use (7.29), (7.30), and (7.32) to find $A = 50$, $\beta = 4.53351$, and $\Delta\Omega$, where

$$\Delta\Omega_{\text{lower}} = \frac{2 \cdot \pi \cdot (4{,}000 - 2{,}000)}{20{,}000} = 0.6263185 \quad \text{rad/sec}$$

$$\Delta\Omega_{\text{upper}} = \frac{2 \cdot \pi \cdot (8{,}000 - 5{,}000)}{20{,}000} = 0.9424778 \quad \text{rad/sec}$$

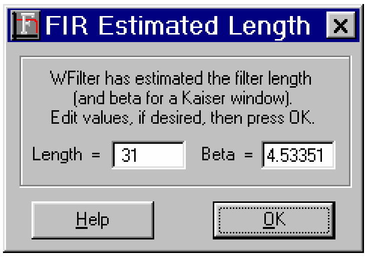
Equation (7.31) can then be used to estimate the smallest odd filter length as $N = 31$. Equation (7.17) can be used to determine the ideal filter coefficients after finding the values of $\tau = 15$ and calculating Ω_{c1} and Ω_{c2}, as shown below.

$$\Omega_{c1} = \frac{2 \cdot \pi \cdot (2{,}000 + 4{,}000)}{2 \cdot 20{,}000} = 0.942478 \quad \text{rad/sec}$$

$$\Omega_{c2} = \frac{2 \cdot \pi \cdot (5{,}000 + 8{,}000)}{2 \cdot 20{,}000} = 2.042035 \quad \text{rad/sec}$$

The Kaiser window coefficients can be determined by using (7.28). The final coefficients can be calculated by multiplying the ideal coefficients by the respective window coefficients and shifting all coefficient indices by a value of $M = 15$.

Rather than perform all of the numerical calculations by hand, we can use WFilter to finish the design of this filter. One addition to the design process for FIR filters is the indication of the filter length and an option to change the length if desired. Since the determination of filter length is not an exact calculation, this option allows the user to increase or decrease the length as necessary to satisfy the design. Figure 7.12 shows the ***FIR Estimated Length*** dialog box, which includes an estimate of the filter length and the value of β.

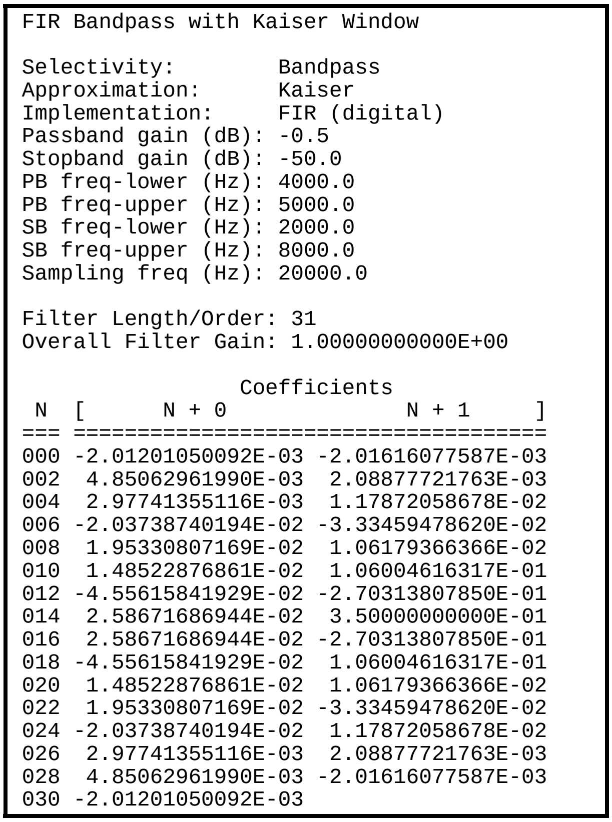


WFilter has estimated the filter length (and beta for a Kaiser window). Edit values, if desired, then press OK.

Length = 31 Beta = 4.53351

Figure 7.12 FIR Estimated Length dialog box.

Figure 7.13 shows the coefficient screen with the final filter coefficients displayed. They are displayed in causal form, and are symmetric about the center value. The magnitude response of the filter is shown in Figure 7.14 and verifies that the specifications have been satisfied.



```
FIR Bandpass with Kaiser Window

Selectivity:          Bandpass
Approximation:        Kaiser
Implementation:       FIR (digital)
Passband gain (dB):   -0.5
Stopband gain (dB):   -50.0
PB freq-lower (Hz):   4000.0
PB freq-upper (Hz):   5000.0
SB freq-lower (Hz):   2000.0
SB freq-upper (Hz):   8000.0
Sampling freq (Hz):   20000.0

Filter Length/Order: 31
Overall Filter Gain: 1.00000000000E+00

                    Coefficients
 N  [      N + 0              N + 1      ]
=== =====================================
000 -2.01201050092E-03 -2.01616077587E-03
002  4.85062961990E-03  2.08877721763E-03
004  2.97741355116E-03  1.17872058678E-02
006 -2.03738740194E-02 -3.33459478620E-02
008  1.95330807169E-02  1.06179366366E-02
010  1.48522876861E-02  1.06004616317E-01
012 -4.55615841929E-02 -2.70313807850E-01
014  2.58671686944E-02  3.50000000000E-01
016  2.58671686944E-02 -2.70313807850E-01
018 -4.55615841929E-02  1.06004616317E-01
020  1.48522876861E-02  1.06179366366E-02
022  1.95330807169E-02 -3.33459478620E-02
024 -2.03738740194E-02  1.17872058678E-02
026  2.97741355116E-03  2.08877721763E-03
028  4.85062961990E-03 -2.01616077587E-03
030 -2.01201050092E-03
```

Figure 7.13 Coefficient values for Example 7.3.

7.3 PARKS-MCCLELLAN OPTIMIZATION PROCEDURE

As we can see in Figure 7.14 for the filter using the Kaiser window, the stopband has ripple that generally decreases. In fact, if the passband characteristic were magnified, we would see ripple there as well. Both the passband and stopband ripple (error) tend to be larger near the transition bands and then taper off as the response moves away from the band edge. This type of response is not optimum. An optimum filter would have ripple in the passband and stopband with a constant maximum magnitude. The error would still be present but would be distributed equally throughout the bands.

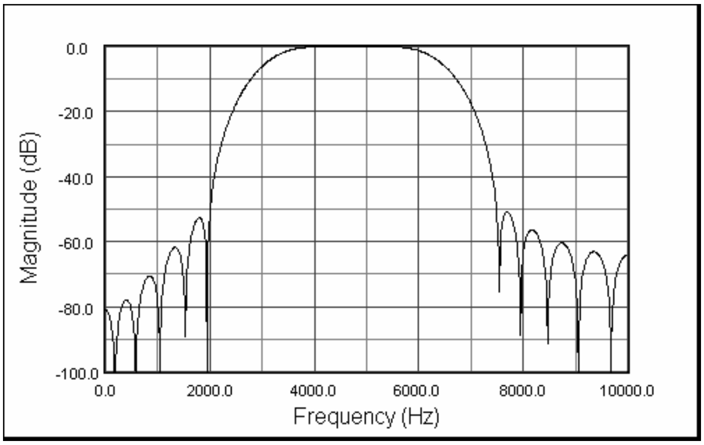


Figure 7.14 Magnitude response for Example 7.3.

In this section, we discuss a method for designing FIR filters with this characteristic. The Parks-McClellan algorithm, as it is generally known, was first presented over 20 years ago. Although the basic procedure has remained the same, a number of implementation techniques have changed. A detailed description of the Parks-McClellan (PM) algorithm is beyond the scope of this text, but a general overview of the procedure is appropriate. In addition, a review of the commented filter design code (as introduced in the next section) provides further implementation details. (Several of the texts listed in the digital filter design section of Appendix A provide more detailed descriptions of the PM algorithm, with the text by Antoniou providing a particularly detailed description.)

7.3.1 Description of the Problem

A primary component of the PM algorithm is a technique called the Remez exchange algorithm. But before we can use the power of the Remez algorithm to optimize our FIR filter coefficients, we must redefine our problem in such a way that the solution requires the minimization of an error function. To that end, we first define the frequency response of an odd-order FIR filter with symmetrical coefficients $h(n)$, as shown in (7.33):

$$H(e^{j\omega}) = \sum_{n=0}^{N-1} h(n) \cdot e^{-j\omega n} = e^{-j\omega M} \sum_{m=0}^{M} c(m) \cdot \cos(m\omega) \tag{7.33}$$

where $M = (N - 1) / 2$ and

$$c(m) = \begin{cases} h(M), & \text{for } m = 0, \\ 2 \cdot h(M - m), & \text{for } m = 1, 2, \ldots, M \end{cases} \tag{7.34}$$

We can then define the summation component of (7.33) as

$$C(\omega) = \sum_{m=0}^{M} c(m) \cdot \cos(m\omega) \tag{7.35}$$

which is used to formulate the error function that will be the object of the minimization. As shown in (7.36), the error function can be described in terms of the desired frequency response $e^{-j\omega M} D(\omega)$, the actual frequency response $e^{-j\omega M} C(\omega)$, and a weighting function $W(\omega)$ that can be used to adjust the amount of error in each filter band:

$$E(\omega) = W(\omega) \cdot \left[D(\omega) - C(\omega)\right] \tag{7.36}$$

The desired frequency response function $D(\omega)$ is usually defined as being 1 within the passband of the filter and 0 within the stopband, although other values can be assigned. The weighting function $W(\omega)$ can be defined equivalently throughout the filter band, or it can be assigned a value of 1 within the passband and 10 within the stopband if a smaller error value δ is desired in the stopband. This result occurs because the minimization algorithm will produce equal amounts of error throughout the defined frequency range, and since the stopband error has been artificially increased by 10, the actual error will be 10 times smaller.

The optimum error function will produce variations within the passband and stopband similar to those shown in Figure 7.2 (except that all ripple will be of the same magnitude). The actual error function will alternate between positive and negative δ values because of the summation of cosine functions. If we pick a set of frequencies ($x = M + 1$) at which the extremes of the error occur, (7.36) can be written as

$$E(\omega_i) = W(\omega_i)\cdot\left[D(\omega_i) - C(\omega_i)\right] = (-1)^i\,\delta,$$
$$\text{for } i = 0, 1, \ldots, x \tag{7.37}$$

Equation (7.37) can be expanded into a matrix equation by considering these x+1 frequencies, which are typically called extremals and play a crucial role in the optimization process.

$$\begin{bmatrix} 1 & \cos\omega_0 & \cdots & \cos M\omega_0 & \dfrac{1}{W(\omega_0)} \\ 1 & \cos\omega_1 & \cdots & \cos M\omega_1 & \dfrac{-1}{W(\omega_1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & \cos\omega_x & \cdots & \cos M\omega_x & \dfrac{(-1)^x}{W(\omega_x)} \end{bmatrix} \cdot \begin{bmatrix} c(0) \\ c(1) \\ \vdots \\ c(M) \\ \delta \end{bmatrix} = \begin{bmatrix} D(\omega_0) \\ D(\omega_1) \\ \vdots \\ D(\omega_x) \end{bmatrix} \tag{7.38}$$

In (7.38), $\omega_0 - \omega_x$ represents the extremal frequencies and δ is the error. With this expression the filter design problem has been set into a form that can be manipulated by the Remez exchange algorithm.

7.3.2 The Remez Exchange Algorithm

The Remez exchange algorithm is a powerful procedure that uses iteration techniques to solve a variety of minimax problems. (A minimax problem is one in which the best solution is the one that minimizes the maximum error that can occur.) Before initiating the process, a set of discrete frequency points is defined for the passband and stopband of the filter. (Transition bands are excluded.) This dense grid of frequencies is used to represent the continuous frequency spectrum. Extremal frequencies will then be located at particular grid frequencies as determined by the algorithm. The basic steps of the method as it is applied to our filter design problem are shown below.

Remez Exchange Algorithm

I. Make an initial guess as to the location of x + 1 extremal frequencies, including an extremal at each band edge.

II. Using the extremal frequencies, estimate the actual frequency response by using the Lagrange interpolation formula.

III. Locate the points in the frequency response where maximums occur and determine the error at those points.

IV. Ignore all new extremals beyond the number initially set in I.

V. If the difference between the maximum and minimum error at the remaining extremal is small enough, continue to VI. Otherwise return to II using the retained extremals.

VI. Estimate the final frequency response and determine the *c*(*m*) values from it. Then determine the impulse response coefficients.

Each step in the procedure can be implemented in a variety of ways. These variations can produce differences in the speed of executing the algorithm, but usually little difference in accuracy is noticed. The simplest method of implementing step I is to assign the $x + 1$ extremal frequencies such that they are equally spaced throughout the bands of interest. Extremals are usually placed at all band edges that are adjacent to transition bands. The initial band and final band may not have extremals located at their terminal edges. The barycentric form of the Lagrange interpolation formula (as described in the mathematical references in Appendix A) is then used to determine the frequency response on the dense grid of frequencies. This method is much more efficient and accurate than the alternative method of finding the *c*(*m*) values in (7.37) by matrix inversion. Once the frequency response has been determined, the true extrema can be located and the error at these locations calculated. (Various methods can be used to locate the extrema, usually differing in speed and complexity.) These new frequency points will be used as the new extrema in the next iteration.

It is not unusual to find more extrema in the frequency response than will be needed to characterize the final frequency response. Therefore, some means is necessary to reduce the number of retained frequencies to $x + 1$. Again, there are variations on this procedure, but the general consensus is to retain the extremals that produce the largest error. In step V, we check the difference between the largest and smallest error produced at the retained extremals. By using this value as a progress indicator, we can set some threshold to indicate when the procedure has produced the required level of optimization. If the differences between the minimum and maximum errors have not been reduced enough, the algorithm continues from step II. When the optimization procedure has reached the desired threshold, the extremal frequencies can be used to determine the *c*(*m*) values in (7.37) and therefore the impulse response coefficients *h*(*n*) from (7.34).

7.3.3 Using the Parks-McClellan Algorithm

The general algorithm has great flexibility in designing any of the four types of FIR filters discussed earlier. The code that is included with this text will design lowpass, highpass, bandpass, and bandstop type 1 filters (with an odd number of symmetrical coefficients). The code is written so that other filter types can be implemented by adding to the program structure. In order to use the general algorithm, we must first convert our filter specifications into those needed by the algorithm. This amounts to converting gain requirements for decibels to absolute error and some redefinition of frequencies.

As in the Kaiser window case, an empirical formulation of the required length of an FIR filter designed using the PM algorithm has been developed, as shown in (7.39). Although somewhat extensive in its presentation, it does provide an accurate estimate of the required length.

$$N = \frac{K_1 - K_2 \cdot \Delta f^2}{\Delta f} + 1 \tag{7.39}$$

where

$$\begin{aligned} K_1 &= [0.005309 \cdot (\log \delta_p)^2 + 0.07114 \cdot \log \delta_p - 0.4761] \cdot \log \delta_s \\ &\quad - [0.00266 \cdot (\log \delta_p)^2 + 0.5941 \cdot \log \delta_p + 0.42781] \end{aligned} \tag{7.40}$$

$$K2 = 0.51244 \cdot (\log \delta_p - \log \delta_s) + 11.012 \tag{7.41}$$

$$\Delta f = (f_{\text{stop}} - f_{\text{pass}}) / f_s \tag{7.42}$$

<u>Example 7.4 Determining Parks-McClellan Coefficients for FIR Filter</u>

Problem: Determine the impulse response coefficients for a bandpass filter using the same specifications as in Example 7.3 (as indicated below), except use the Parks-McClellan algorithm for coefficient determination.

f_{pass1} = 4 kHz, f_{pass2} = 5 kHz, f_{stop1} = 2 kHz, f_{stop2} = 8 kHz,
a_{pass} = –0.5 dB, $a_{\text{stop1}} = a_{\text{stop2}}$ = –50 dB, and f_{samp} = 20 kHz

Solution: We will use the WFilter program for this example. The same input parameters as in the previous example are specified, except the approximation type has been changed to Parks-McClellan. Using the specified parameters, the first design attempt resulted in an estimated length of 19, which produced a filter with passband edge gains of –0.54 dB and stopband edge gains of –49.38 dB. These values are certainly very close to the design specifications and might be acceptable in many designs. However, for comparison purposes, the filter was redesigned using a filter length of 21 and produced passband gains of –0.23 dB and stopband gains of –56.70 dB. The resulting coefficients and frequency response curve are shown in Figures 7.15 and 7.16.

We should notice the equal ripple in the stopbands for the PM filter and the fact that it is implemented in one-third fewer coefficients than the Kaiser filter. As a comparison to IIR filters, a sixth-order elliptic or an eighth-order Butterworth filter would be required to satisfy the same specifications, but without linear phase.

```
FIR Bandpass using Parks-McClellan Procedure

Selectivity:          Bandpass
Approximation:        Parks-McClellan
Implementation:       FIR (digital)
Passband gain (dB):   -0.5
Stopband gain (dB):   -50.0
PB freq-lower (Hz):   4000.0
PB freq-upper (Hz):   5000.0
SB freq-lower (Hz):   2000.0
SB freq-upper (Hz):   8000.0
Sampling freq (Hz):   20000.0

Filter Length/Order: 21
Overall Filter Gain: 1.00000000000E+00

                 Coefficients
 N  [      N + 0              N + 1     ]
=== =====================================
000  1.25270567042E-02  1.19473087473E-03
002 -3.33680410407E-02 -4.33317885804E-03
004  1.22816612467E-02  8.32245424391E-03
006  1.02738836518E-01 -8.97234696493E-03
008 -2.68080507538E-01  4.01012968419E-03
010  3.48822141957E-01  4.01012968419E-03
012 -2.68080507538E-01 -8.97234696493E-03
014  1.02738836518E-01  8.32245424391E-03
016  1.22816612467E-02 -4.33317885804E-03
018 -3.33680410407E-02  1.19473087473E-03
020  1.25270567042E-02
```

Figure 7.15 Coefficient values for Example 7.4.

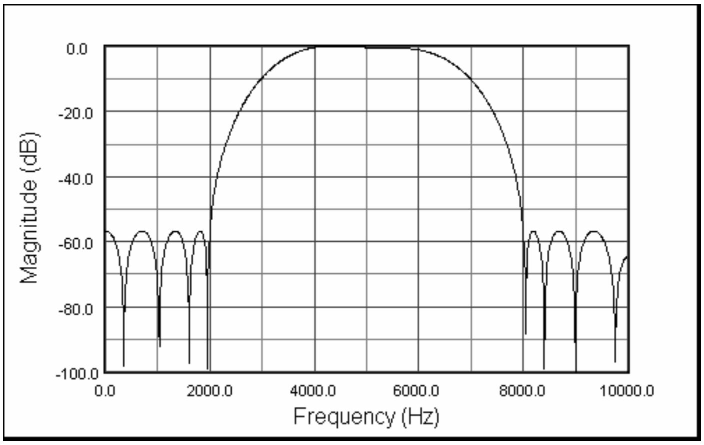


Figure 7.16 Magnitude response for Example 7.4.

7.3.4 Limitations of the Parks-McClellan Algorithm

The Parks-McClellan algorithm is certainly an attractive FIR filter design method, but it does have certain limitations in comparison to window-based design methods. The Kaiser window design is basically a one-pass system, although some variation of filter length may be necessary to attain the desired specification (as is also the case for the Parks-McClellan procedure). The computational intensity of the PM method is far greater than for any of the window methods, but computational power is also more readily available than it was 10 years ago. Probably the biggest problem with the PM algorithm is that it does not *always* lead to a solution. In some cases, the iteration sequence will not converge, which makes it necessary to place an upper limit on the number of iterations allowed. The algorithm can be modified to allow the frequency grid to be made more dense and to initiate the algorithm again if this happens. Nonetheless, there will be occurrences when the problem statement will need to be redefined in order to attain convergence. For example, if the two stopbands of a bandpass filter are not of approximately equal size, one can be artificially reduced to help the convergence process. Occasionally, the error constraints on the filter must be adjusted to allow more freedom in the optimization process. Even if the process converges, the frequency response of the resulting filter must be checked carefully. Since no requirements are placed on the frequency within the transition bands, strange results can sometimes occur.

7.4 C CODE FOR FIR FREQUENCY RESPONSE CALCULATION

Our last task is to determine the frequency response of the filter. As determined in Chapter 5, the frequency response of a digital filter can be determined from the transfer function, as shown in (7.43):

$$H(e^{j\Omega}) = H(z)\big|_{z=e^{j\Omega}} \tag{7.43}$$

For a causal FIR filter, the transfer function can be described in (7.44), which then leads to the description for the frequency response shown in (7.45):

$$H(z) = \sum_{k=0}^{N-1} h(k)\cdot z^{-k} \tag{7.44}$$

$$H(e^{j\Omega}) = \sum_{k=0}^{N-1} h(k)\cdot e^{-jk\Omega} \tag{7.45}$$

Equation (7.45) can also be expressed as a sum of the real and imaginary portions of the exponential as in (7.46). The `Calc_DigFIR_Resp` function implements (7.46) directly, as shown in Listing 7.1.

$$H(e^{j\Omega}) = \sum_{k=0}^{N-1} h(k)\cdot\cos(k\Omega) + j\sum_{k=0}^{N-1} h(k)\cdot\sin(k\Omega) \tag{7.46}$$

```
/*======================================================
  Calc_DigFIR_Resp() - calcs response for FIR filters
  Prototype:  int Calc_DigFIR_Resp(Filt_Params *FP,
                                   Resp_Params *RP);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
              RP - ptr to struct holding respon params
======================================================*/
int Calc_DigFIR_Resp(Filt_Params *FP,Resp_Params *RP)
{ int     f,i;          /*  loop counters */
  double  rad2deg,      /*  rad to deg conversion */
          omega,i_omega,/*  radian freq and incrmnt */
          mag,          /*  magnitude of freq resp */
          rea,img;      /*  real and imag part */
  rad2deg = 180.0 / PI; /* set rad2deg */
  /*  Loop through each of the frequencies */
  for(f = 0 ;f < RP->tot_pts; f++)
  { /* Initialize magna and angle */
    RP->magna[f] = FP->gain;
    RP->angle[f] = 0.0;
    /* Pre calc adjusted omega, rea and img */
    omega = PI2 * RP->freq[f] / FP->fsamp;
    rea = 0.0; img = 0.0;
    /* Loop through all the coefs */
    for(i = 0 ;i < FP->order; i++)
    { i_omega = i * omega;
      rea += FP->acoefs[i] * cos(i_omega);
      img += FP->acoefs[i] * sin(i_omega);
    }
    /* Calc final result and conv to degrees */
    mag = sqrt(rea*rea + img*img);
    RP->magna[f] *= mag;
    /*  Guard against atan(0,0) */
    if(mag > 0)
    { RP->angle[f] += atan2(img,rea);}
    RP->angle[f] *= rad2deg;
  }
  /*  Convert magnitude response to dB if indicated */
  if(RP->mag_axis == LOG)
  { for(f = 0 ;f < RP->tot_pts; f++)
    { /* Handle very small numbers */
      if(RP->magna[f] < ZERO)
      { RP->magna[f] = ZERO;}
      RP->magna[f] = 20 * log10(RP->magna[f]);
    }
  }
  return ERR_NONE;
}
```

Listing 7.1 `Calc_DigFIR_Resp` function.

The magnitude variable is initialized to the value of the gain constant, and the angle variable is set to zero. Then, the response at each frequency is determined by evaluating the effect of each filter coefficient. The angle is converted to degrees, and after all calculations have been made, the magnitude is converted to decibels if the user requested that format.

7.5 CONCLUSION

By completing this chapter, we have completed the material on digital filter design. We investigated two of the most popular methods of FIR filter design: the Fourier series method using window functions and the Parks-McClellan optimization method. We can also determine and display the magnitude and phase response of the FIR filters we design. In the next chapter we will investigate the implementation of both the FIR and IIR digital filters we have designed. For those interested in the C code for FIR filter design, please refer to Appendix H.

Chapter 8

Digital Filter Implementation Using C

In the previous three chapters we discussed the nature of digital filter design. We are now ready to discuss the implementation of these digital filters. We begin this chapter with a discussion of several important issues in digital filter selection and implementation. These issues include the differences between real-time and nonreal-time implementation, as well as the effects of finite precision representation of input signals and filter coefficients. Then, we discuss the C code for implementing IIR and FIR filters. Efficient algorithms will be developed to increase the speed of execution. Each filter type will use a different technique appropriate to the specific filter's representation. Finally, we conclude with a discussion of the format for a popular sound file on the PC. We will consider how we can use sound files to investigate the characteristics of the filters we have designed.

8.1 DIGITAL FILTER IMPLEMENTATION ISSUES

The first decision to make when designing a system with a digital filter is whether an IIR or FIR filter should be used. Some of the advantages and disadvantages of each type have been discussed in the previous two chapters, so we will summarize those points here. First, and foremost, the correct filter type must be determined by the requirements of the application. IIR (recursive) filters have the advantages of providing higher selectivity for a particular order and a closed form design technique that doesn't require iteration. The design technique also provides for the rather precise solution to the specifications of gain and edge frequencies. However, IIR filters also have the disadvantages of nonlinear phase characteristics and possible instability due to poor implementation. FIR (nonrecursive) filters, on the other hand, can provide a linear phase response (constant group delay) that is important for data transmission and high-quality audio systems. Also, they are always stable because they are implemented using an all-zero transfer function. Since no poles can fall outside the unit circle, the filter will always be stable. But because of this, the order of the filter is much higher than the IIR filter, which has a comparable magnitude response. This higher order leads to longer processing times and larger memory requirements. In addition, FIR filters must be designed using an iterative method since the required filter length to satisfy a given filter specification can only be estimated.

Therefore, the filter designer must weigh the requirements placed on the digital filter. If great importance is placed on magnitude response with much less importance on phase response, then an IIR filter would seem the better choice. If phase response is far more important than magnitude response, then an FIR filter is in order. If both magnitude and phase response seem to be of equal concern, then the processing time constraints and memory requirements must be considered. The FIR filter, even when designed using the Parks-McClellan method, will require more processing time and more memory to implement, but always will be stable. If all else fails, both an FIR and IIR filter (with some phase correction) can be designed to meet the specifications and they both can be tested to evaluate the results.

Once the choice of filter type has been made, there are still a number of decisions to make. For example, is the system to operate in real-time or can it be a nonreal-time system? A real-time system is one in which input samples are provided to the digital filter and must be processed to provide an output sample before the next input sample arrives. Obviously, this puts a very precise time constraint on the amount of processing available to the system. The higher the sampling frequency, the less time is available for processing. On the other hand, some systems are afforded the luxury of being able to operate in nonreal time. For example, signals can be recorded on tape or other media and processed at a later time. In this type of system, extensive processing can take place because there is no fixed time interval that marks the end of the processing time. In nonreal-time applications, high-precision floating-point representation for the coefficients and signal can usually be used since speed is not the critical factor. However, the majority of digital filter applications will be real-time applications. In these applications there will inevitably be a battle to obtain the highest accuracy, the fastest speed, and the lowest-cost system. One of the first decisions to make is whether the system will be a fixed-point or a floating-point system. This deals not only with the input and output signal streams, but also the representation of the coefficients and intermediate results within the processing unit.

8.1.1 Input and Output Signal Representation

Fixed point systems represent a large market in today’s digital signal processing arena. Many analog-to-digital (A/D) converters are available to provide output in a fixed-point representation. Although many input and output digital signals use fixed-point representation, there are several ways in which the same signals can be interpreted. Most people are familiar with the base 10 system that humans use, but the digital computers of the world use a binary or base 2 system. In that system, each digit represents a multiplier of a power of 2 just as each digit in the decimal or base 10 system represents a multiplier of a power of 10. Let's start with an 8-bit binary number shown below:

$$10011000_2 = 1 \cdot 2^7 + 1 \cdot 2^4 + 1 \cdot 2^3 = 152_{10} \tag{8.1}$$

This binary number is equivalent to 152_{10} only when we assume that the number is unsigned (represents only positive numbers). If we had considered the binary number as signed, the leftmost 1 would have indicated a negative sign, and the value would have been interpreted differently. Using two's complement arithmetic, we can determine the value of the number in (8.1) by first negating every digit in the number and then adding one to the result. That value is then considered a negative value, as shown in (8.2):

$$10011000_2 = -(01100111_2 + 1) = -01101000_2 = -104_{10} \tag{8.2}$$

We can even interpret the original binary number in a different way if we assume that the placement of the binary point (equivalent to the decimal point) is located at a position other than to the right of the right-most digit. For example, if we place the binary point in the middle of the eight binary digits, the number takes on another value entirely. (In the representation shown below, it is considered an unsigned number, but it could just as well be considered a signed number as described before.)

$$1001.1000_2 = 1 \cdot 2^3 + 1 \cdot 2^0 + 1 \cdot 2^{-1} = 9.5_{10} \tag{8.3}$$

There are advantages and disadvantages to each of the binary representations. The use of signed numbers is required in most digital filters, and the two's complement representation provides easier methods for addition and subtraction, but multiplication requires special consideration. The signed fractional arithmetic illustrated in (8.3) provides great efficiency in multiplication even though addition and subtraction do require some special steps. By using fractional notation, as most commercial digital signal processing (DSP) chips do, we can guarantee that the result of a multiplication will still be less than one and therefore cause no overflow. However, the addition of two fractional numbers can provide a result larger than one and therefore must be handled carefully. Techniques for dealing with potential errors in calculations are considered further in Section 8.1.3.

Before we leave the area of input and output data representation completely, we must consider the effect of the number of bits per sample on the signal-to-noise ratio (SNR) of a digital system. It is shown in a number of the digital filter design references listed in Appendix A that the SNR for a digital signal processing system will be directly proportional to the number of quantization bits used. To be specific, the equation is

$$\mathrm{SNR}_{\mathrm{dB}} = 6.02 \cdot B + 20 \cdot \log(\sigma_x / A_{\max}) + 10.8 \tag{8.4}$$

In (8.4), B represents the number of bits used to represent the magnitude of the digitized signal, while σ_x and $A_{\max}$ represent the input signal's standard deviation and the quantizer's maximum input signal. Usually, the input signal's amplitude is adjusted such that $\sigma_x/A_{\max}$ is approximately 1/4. If this condition is met, and we assume that the input signal has a Gaussian distribution (which is a valid assumption for many signals), then the quantizer's limits will be exceeded only 0.064% of the time. (If a smaller number is chosen for the ratio, the dynamic range of the system would be sacrificed. If a larger number is chosen, the frequency of overflow would increase.) Using the value of 1/4 gives

$$\mathrm{SNR}_{\mathrm{dB}} = 6.02 \cdot B - 1.2 \tag{8.5}$$

An important realization drawn from this expression is that the SNR can be improved by 6 dB for every bit added to the quantized representation. For example, if a particular filtering application requires a signal-to-noise ratio of 80 dB, then a D/A converter with at least 14 bits of magnitude quantization must be available.

8.1.2 Coefficient Representation

When it comes to the internal processing of data for a digital filter system, there is more of an even mix between fixed-point and floating-point systems. The market has a variety of DSP microprocessor chips from a number of manufacturers. These DSP systems include a wide selection of fixed-point and floating-point systems. The fixed-point systems generally provide higher processing speed at lower cost than do the floating-point systems. However, the floating-point systems provide accuracy that many fixed-point systems cannot achieve. The representation for the coefficients does not have to be the same as the representation selected for the input signal. Even if both are fixed-point numbers, the coefficient representation can use a higher precision representation (more bits). It is up to the digital filter designer to determine the system characteristics such that sufficient accuracy is achieved with adequate processing speed at the lowest possible cost.

The representation of the filter coefficients within the DSP system is of prime concern. Enough accuracy is required in the representation of the filter coefficients (which determine the pole and zero locations) to guarantee that the specifications of the filter are met by the implementation. If the accuracy of the coefficients is compromised, the response of the filter may be severely distorted, and in some cases, the stability of IIR filters can be jeopardized. FIR filters will always be stable because they are represented by transfer functions with all zeros.

However, their frequency response can still be affected by the lack of accuracy of their coefficients. As an example, we consider the case of the Parks-McClellan filter designed in Example 7.4. The coefficients for that example have been truncated to signed 16-bit, 12-bit, and 8-bit numbers as shown in Table 8.1.

Table 8.1

Comparison of Original and Truncated Coefficients

Coefs	*Original*	*16-bit*	*12-bit*	*8-bit*
h(10)	0.348822	0.348822	0.348822	0.348822
h(9), h(11)	0.004010	0.004013	0.004090	0.002747
h(8), h(12)	-0.268081	-0.268076	-0.268049	-0.269170
h(7), h(13)	-0.008972	-0.008974	-0.009032	-0.008240
h(6), h(14)	0.102739	0.102740	0.102755	0.101625
h(5), h(15)	0.008322	0.008325	0.008350	0.008240
h(4), h(16)	0.012282	0.012285	0.012269	0.010987
h(3), h(17)	-0.004333	-0.004333	-0.004260	-0.005493
h(2), h(18)	-0.033368	-0.033363	-0.033400	-0.032960
h(1), h(19)	0.001195	0.001192	0.001193	0.000000
h(0), h(20)	0.012527	0.012530	0.012610	0.013733

Equation (8.6) shows the procedure for determining the truncated values. The calculation within the *Round* function actually indicates the procedure of converting the floating-point coefficient to a signed fraction representation for use in fixed-point processors. The multiplication outside of the *Round* function converts the signed fraction back to a truncated decimal:

$$h_{\text{trunc}}(n) = Round\left[\frac{h(n)\cdot(2^{B-1}-1)}{h(10)}\right]\cdot\frac{h(10)}{(2^{B-1}-1)} \tag{8.6}$$

As we would expect, as the number of bits is reduced, the error in the coefficients increases. The frequency response generated from the 16-bit coefficients was virtually identical to the original response. However, the responses due to the 12-bit and 8-bit coefficients were noticeably degraded, as shown in Figure 8.1. In that figure, the passband response was virtually unchanged in all cases, but the stopband characteristics did not match those of the original coefficients (shown as –56.7 dB). The 12-bit coefficients did produce a response that satisfied the original design specifications of –50 dB. The 8-bit coefficients, however, caused the frequency response to degenerate completely, providing as little as 32 dB of attenuation. The reason for this degradation, of course, is that the truncation has moved the pole and zero locations from their original positions.

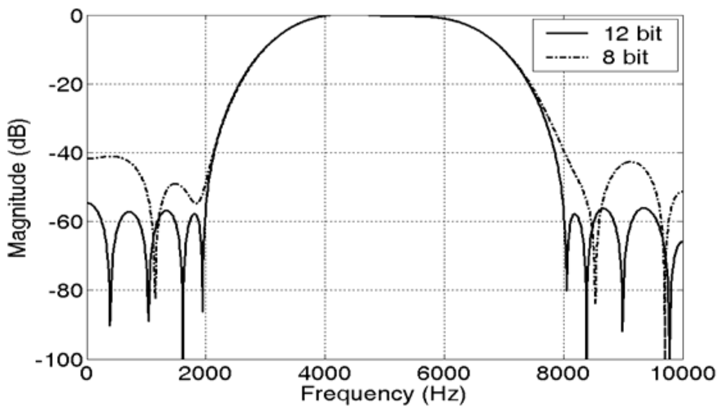


Figure 8.1 FIR filter frequency response using truncated coefficients.

8.1.3 Retaining Accuracy and Stability

There is no effective way to directly relate the accuracy of the coefficients to the degree of degradation of the frequency response. At this time, trial and error techniques are all that can be offered when determining the necessary accuracy of coefficients. However, there are some helpful practices that can be observed when dealing with the implementation of these coefficients to reduce the effects of truncation. The following suggestions relate to IIR filters unless stated otherwise since they have some unique problems due to their recursive nature. The fact that IIR filters are implemented using feedback leads to special problems that the FIR filter does not experience.

Probably the most important implementation rule when dealing with IIR filters is that it is much better to implement the filter as a cascade of quadratic factors (as we have done) than to combine the transfer function into a quotient of high-order polynomials. (Even some experimentation has been done with this idea for FIR filters.) This technique provides better control of the stability of the filter. By quantizing the coefficients that represent the filter, we are actually specifying a fixed number of positions within the unit circle where the poles can be located. The fewer bits used for quantization, the fewer the positions for the poles. In addition, these pole locations are not uniformly distributed within the unit circle. However, by choosing different topologies for the implementation (such as a coupled form of quadratic), more uniformly distributed pole locations can be achieved. Also, it has been shown that those poles which either lie close to the unit circle or to each other are the most critical to represent properly, and therefore may need some special implementation method. Another technique that can be used to reduce errors caused by computations using finite accumulators is to pair poles and zeros located near each other in the same quadratic factor to reduce large fluctuations. In addition, sections with poles closest to the unit circle can be moved to the end of the evaluation process to help eliminate overflows. The digital filter design references in Appendix A provide further information on other structures than the quadratic form, including those for representing poles located extremely close to the unit circle. Antoniou's, Oppenheim/Shafer's (1975), and Proakis/Manolakis's texts provide valuable and detailed material.

Several other points can be made when discussing the coefficients and internal processing of the filter calculations. In both FIR and IIR filter implementation, the final output values are stored in a register that gradually accumulates the value of the final output. This accumulator should normally be allocated as many bits of representation as possible because it controls the ultimate accuracy of the processor. Overflow and underflow are potential problems for the accumulator since it is hard to predict the exact nature of incoming signals. Most present dedicated DSP chips provide some form of scaling that can be applied to the input so that temporary large variations can be accommodated without incurring a great deal of error. This scaling bit can effectively be used as a temporary additional bit of accuracy for accumulated values to prevent overflow. However, if overflow is inevitable, it is better to have a processor that will simply saturate at its maximum level than to allow an overflow that can be interpreted as a swing from positive to negative value.

In recursive systems that use finite precision representations, a troublesome problem called limit cycle oscillation can occur. There are actually two types of limit cycles: overflow and quantization. Overflow limit cycles (also called large-signal limit cycles) are due to the unmanaged overflow of the accumulator during processing. Most overflow limit cycles can be effectively eliminated by the proper use of signal scaling at the input of the system and saturation arithmetic in the accumulator. Many DSP processors include some scaling feature within the processor unit that allows the input signal to be reduced in size. However, a reduction in the size of the input signal also reduces the SNR of the system, although this is usually better than the distortion produced by an overflow. Another useful feature on many DSP systems today is the use of saturation arithmetic within the processing unit. This feature will saturate the value to its positive or negative limit rather than overflow the accumulator. Again, this will result in distortion, but usually less than the overflow would produce. Of course, another way to help control overflow limit cycles is to increase the size of the accumulator, if the processing system allows the accumulator to be larger than the standard coefficient storage size.

The quantization limit cycles (also called small-signal limit cycles) generally result from the handling of quantization within the system and are noticeable when the output should be constant or zero. This problem is the result of the input signal changes being less than the quantization level. Two methods have been developed to combat this type of limit cycle. The first, which may not be a practical alternative, is to increase the number of bits assigned to the representation of the signal values in order to reduce the quantization error. By reducing the quantization error, the limit cycles can either be eliminated altogether, or reduced to a tolerable level. The second method suggests that the products produced by finite precision multiplication be truncated, rather than rounded, as they are accumulated. Other, more detailed, analysis of limit cycles is included in the references.

8.2 C CODE FOR IIR FILTER IMPLEMENTATION

When discussing the implementation of IIR filters, we assume that the filter is described by a set of quadratic coefficients of the form determined in Chapter 7. As we have seen in the previous section, a cascaded sequence of quadratic structures is the recommended method of implementation. The basic quadratic building block is shown in the system diagram of Figure 8.2.

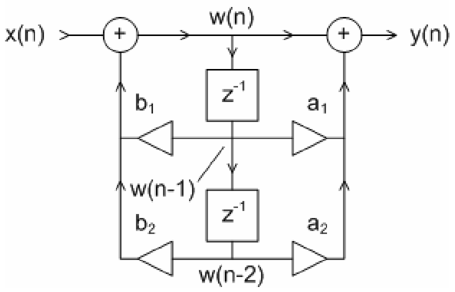


Figure 8.2 System diagram for a single quadratic factor.

We can generate the transfer function for this section by determining the expressions for the intermediate signal *w*(*n*) and the output signal *y*(*n*).

$$w(n) = x(n) + b_1 \cdot w(n-1) + b_2 \cdot w(n-2) \tag{8.7}$$

$$y(n) = w(n) + a_1 \cdot w(n-1) + a_2 \cdot w(n-2) \tag{8.8}$$

These equations can be *z*-transformed to give

$$W(z) = X(z) + b_1 \cdot z^{-1} \cdot W(z) + b_2 \cdot z^{-2} \cdot W(z) \tag{8.9}$$

$$Y(z) = W(z) + a_1 \cdot z^{-1} \cdot W(z) + a_2 \cdot z^{-2} \cdot W(z) \tag{8.10}$$

Equations (8.9) and (8.10) can be rewritten and combined to determine the transfer function for this section of the filter, as shown in (8.11). This formulation matches the quadratic terms we developed for IIR filters. We will be able to match equivalent terms if we recognize two characteristics. First, the a_o coefficient is always one, and second, the *b* coefficients in the system diagram will be the negative of their value in the transfer function equation.

$$H(z) = \frac{Y(z)}{X(z)} = \frac{(1 + a_1 \cdot z^{-1} + a_2 \cdot z^{-2})}{(1 - b_1 \cdot z^{-1} - b_2 \cdot z^{-2})} \tag{8.11}$$

As an example, consider the fourth-order transfer function for a Chebyshev highpass filter shown in (8.12). This filter can be implemented by using the system diagram shown in Figure 8.3. As we see in the diagram, the input signal is first multiplied by the gain constant and is then processed through two quadratic factors. The multiplication by the gain constant could occur at the end of the process or be distributed throughout the diagram as well. Notice the sign difference on the *b* coefficients between the transfer function of (8.12) and the system diagram of Figure 8.3. (Of course, the coefficients will be represented with higher precision than indicated here.)

$$H_{C4}(z) = \frac{0.201 \cdot (1 - 2 \cdot z^{-1} + 1 \cdot z^{-2}) \cdot (1 - 2 \cdot z^{-1} + 1 \cdot z^{-2})}{(1 - 1.047 \cdot z^{-1} + 0.795 \cdot z^{-2}) \cdot (1 - 0.0448 \cdot z^{-1} + 0.222 \cdot z^{-2})} \tag{8.12}$$

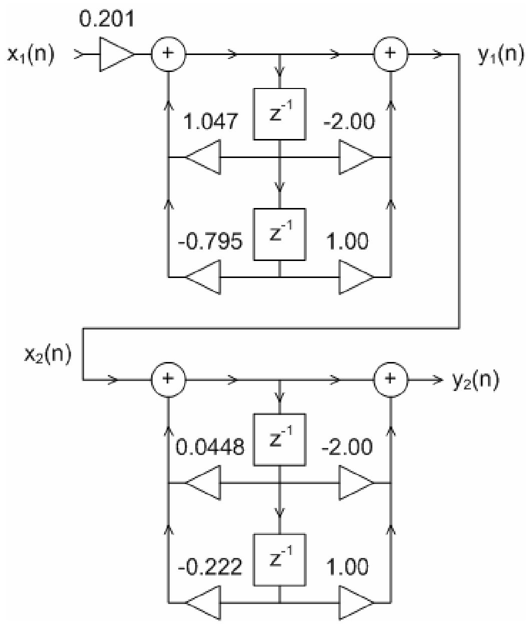


Figure 8.3 System diagram for fourth-order IIR filter.

After determining the system diagram for a filter, we can use the diagram as a guide to implementing the filter. For every quadratic factor, we can calculate an intermediate signal $w(n)$ and an output signal $y(n)$. Besides $x(n)$, $w(n)$, and $y(n)$, every quadratic section also requires the values $w(n - 1)$ and $w(n - 2)$ (as shown in Figure 8.2) to be retained. We can rewrite (8.7) and (8.8) as

$$w = x + b_1 \cdot m_1 + b_2 \cdot m_2 \tag{8.13}$$

$$y = w + a_1 \cdot m_1 + a_2 \cdot m_2 \tag{8.14}$$

where we have defined

$$m_1 = w(n-1) \tag{8.15}$$

$$m_2 = w(n-2) \tag{8.16}$$

The names m_1 and m_2 are picked to reflect the fact that these values are the memory states of the quadratic factor. Each quadratic structure must keep track of the previous values that have been present in the structure.

The IIR filtering process can be implemented by first multiplying the input signal by the gain and then implementing (8.13) to (8.16) for each of the quadratics. Then, before progressing to the evaluation of the next quadratic factor, the values of m_1 and m_2 are updated. A section of code that will implement this process is shown in Listing 8.1. In the listing, it is assumed that there are `m1` and `m2` arrays with sizes equal to the number of quadratics, and that the `a` and `b` coefficients for the filter are stored in the usual manner. (That is, the three coefficients a_0, a_1, and a_2 for the first quadratic are followed by the three coefficients for the second quadratic, and so on. The same organization is assumed for the b coefficients.) Note that the a_0 and b_0 coefficients are not used since they are assumed to be one. We have also substituted the output variable `o` for the input variable x to let the output value accumulate through several quadratic sections.

```
o = x * gain;
for(j = 0; j < numb_quads ;j++)
{ jj = j*3;
  w = o - m1[j] * b[jj+1] - m2[j] * b[jj+2];
  o = w + m1[j] * a[jj+1] + m2[j] * a[jj+2];
  m2[j] = m1[j];
  m1[j] = w;
}
```

Listing 8.1 Segment of code for IIR filter implementation.

Although the algorithm in Listing 8.1 will make the correct computations, it could be slow because of all of the index calculations that must be made. (Actually, compilers will differ in terms of how much optimization can be made with speed-critical code such as we are discussing.) The speed of this loop can usually be increased by using pointers to the coefficients and memory values. In order to take advantage of this pointer efficiency we must store the filter coefficients in an orderly manner, which will allow them to be accessed sequentially in the exact order that they are needed. We can define a new array `C` that will store all of the needed coefficients as well as the filter gain constant. The first element in the array will be the gain constant followed by the coefficients b_1, b_2, a_1, and a_2 for each quadratic factor. The structure of the `C` array then has the following form:

```
C[0] = gain            C[5] = b1 (quad 2)
C[1] = b1 (quad 1)     C[6] = b2 (quad 2)
C[2] = b2 (quad 1)     C[7] = a1 (quad 2)
C[3] = a1 (quad 1)     C[8] = a2 (quad 2)
C[4] = a2 (quad 1)     ...
```

In addition, an `M` array must be created to store the memory states of the IIR filter. The size of the array is equal to twice the number of quadratic factors with the m_1 states stored in the first half of the array and the m_2 states in the last half. Initially, all of the memory states are set to zero (unless we know some predefined state exists for the filter). Thus, the `M` array has the following structure where N is the number of quadratics:

```
M[0] = m1 (quad 1)     M[N] = m2 (quad 1)
M[1] = m1 (quad 2)     M[N + 1] = m2 (quad 2)
M[2] = m1 (quad 3)     M[N + 2] = m2 (quad 3)
...                              ...
```

Listing 8.2 shows the `Dig_IIR_Filter` function used in the DIGITAL program discussed later in this chapter. (All of the functions discussed in this chapter can be found in the \DIGITAL\DIGITAL.C module on the software disc that accompanies this text.) The function takes as arguments pointers to arrays of input values (`X`), output values (`Y`), memory states (`M`), and coefficient values (`C`), as well as the number of quadratic factors and number of values in the input and output arrays. Notice that since we will be progressing through the `X`, `Y`, `C`, and `M` arrays, we have defined indexing pointers `x`, `y`, `c`, and `m,` which can take on changing values. (*Never* use the array name itself as an index or the address of the array will be lost.) The notation may look a little foreign so let's take a look at the code on a line-by-line basis. (The line numbers shown are to help with this discussion and are not part of the normal code.) The process of using pointers is very efficient for computational purposes because it fits the nature of the computer. However, it can be a bit confusing to follow. Therefore, in order to provide a more descriptive analysis of the IIR filter code, the code has been annotated with the status of primary arrays and variables. For this illustration, it is assumed that there are two quadratic factors and sample values have been entered for the `C` and `M` arrays. At each step of the process, the particular values to which `c`, `m1`, or `m2` are pointing within the arrays are shown in bold. Only the first time through the inner loop is illustrated, but we can see from this how the pointers progress through the arrays, and how the memory state array is changed.

In lines 1 and 2 of Listing 8.2, the addresses of the input and output arrays are transferred to temporary variables `x` and `y` that can change without affecting the addresses stored in `X` and `Y`. Then, line 4 begins a `for` loop to process every value in the input array. Lines 6 to 8 set up pointers to the memory states and coefficient arrays. The `m1` pointer is set to start at the beginning of the `M` array where the m_1 states are stored, while the `m2` pointer is set midway through the `M` array since the m_2 states are stored in the second half of the array. The initial value of the output, indicated by `o`, is calculated by multiplying the input value by the first value in the coefficient array, which is the gain constant.

```
/*======================================================
  Dig_IIR_Filter() - filters input array using IIR
              coefs and mem values to generate output
  Prototype:  void Dig_IIR_Filter(int *X,int *Y,
            double *M,double *C,int numb_quads,int N);
  Return:     error value.
  Arguments:  X - ptr to input array
              Y - ptr to output array
              M - ptr to memory array
              C - ptr to coefs array
              numb_quads - number of quadratics
              N - number of values in array
=====================================================*/
void Dig_IIR_Filter(int *X,int *Y,double *M,double *C,
                                 int numb_quads,int N)
{ int     *x,*y,    /*  ptrs to in/out arrays */
          i,j;      /*  loop counters */
  double  *c,*m1,*m2, /*  ptrs to coef/memory arrays*/
          w,o;      /*  intermed & output values */
  /*  Make copies of input and output pointers */
01  x = X;
02  y = Y;
                   x: 3,4,5,6,7,...
                   y: 0,0,0,0,0,...
03  /*  Start loop for number of data values */
04  for(i = 0; i < N ;i++)
05  { /*  Make copies of pointers and start calcs */
06    m1 = M;
07    m2 = M + numb_quads;
08    c = C;
                   C: 0.20,1.0,-0.80,-2.0,-1.0,0.05,-0.20,-2.0,-1.0
                   M: 1.0,3.0,2.0,4.0
09    o = *x++ * *c++;
                   C: 0.20,1.0,-0.80,-2.0,-1.0,0.05,-0.20,-2.0,-1.0
                   x: 3,4,5,6,7,...
                   o: 0.60
```

Listing 8.2 `Dig_IIR_Filter` function.

```
      /*  Start loop for number of quad factors */
      for(j = 0; j < numb_quads ;j++)
      { w = o - *m1 * *c++;
                   C: 0.20,1.0,-0.80,-2.0,-1.0,0.05,-0.20,-2.0,-1.0
                   M: 1.0,3.0,2.0,4.0
                   w: -0.40
        w -= *m2 * *c++;
                   C: 0.20,1.0,-0.80,-2.0,-1.0,0.05,-0.20,-2.0,-1.0
                   M: 1.0,3.0,2.0,4.0
                   w: 1.20
        o = w + *m1 * *c++;
                   C: 0.20,1.0,-0.80,-2.0,-1.0,0.05,-0.20,-2.0,-1.0
                   M: 1.0,3.0,2.0,4.0
                   o: -0.80
        o += *m2 * *c++;
                   C: 0.20,1.0,-0.80,-2.0,-1.0,0.05,-0.20,-2.0,-1.0
                   M: 1.0,3.0,2.0,4.0
                   o: -2.8
        *m2++ = *m1;
                   M: 1.0,3.0,1.0,4.0
        *m1++ = w;
                   M: 1.2,3.0,1.0,4.0
      }
      /*  Convert output to int and store */
      *y++ = (int)ceil(o-0.5);
                   y: -3,0,0,0,0,...
    }
}
```

Listing 8.2 Continued.

Notice that the method used to access the gain constant is by using the `*c` notation, which can be read as "the value at which *c* is pointing." Since `c` has just been initialized to the start of the `C` array, and since the first element in `C` is the gain constant, then `c` is pointing at the gain constant. The "++" notation after `c` instructs the computer to increment the value of `c` by one *after* the equation has been evaluated. (This is referred to as post-incrementing as opposed to `++c`, which is called preincrementing.) At this point, we are performing pointer arithmetic, which is a special type of arithmetic. Remember that the variable `c` (and `C`) contains a number that is an address of where coefficients are stored in memory. When we increment a pointer we are incrementing an address and therefore must do so in a meaningful way. In this case, the coefficients are stored as `double`s, which take up 8 bytes of memory on a PC. It would be meaningless to increment the address of an array containing `double`s by 1 byte, or 2, or anything but 8 bytes. Therefore, when we tell the machine to increment `c`, it increments the number stored in `c` by 8. (This is what is meant by special arithmetic.) The compiler keeps track of how to increment pointer variables based on how the variables are initially declared. For example, if the variables are `double`, the increment is 8; if they are `int`, the increment is 2.

Now, returning to our discussion of Listing 8.2, we next enter a `for` loop (line 11) used to calculate the effects of each quadratic factor on the ultimate output value. Lines 12–13 compute the value of `w` and increment the pointer in the `C` array as each value is used. Lines 14–15 then make the calculations representing the right half of Figure 8.2 to determine the output value. The pointer `c` is now pointing to the `b` coefficients for the next quadratic section. Lines 16–17 update the values of the memory states and increment `m1` and `m2` to point to the memory states of the next quadratic section. The final step in the determination of the output value is the conversion of the floating-point value of `o` to the fixed point value of `*y`. The method used in converting the floating point value to an integer value insures that both positive and negative values will be rounded correctly. The pointer `y` is then incremented and the process repeats for the next value in the input array. We should note that by the end of the outer `for` loop, the `c, m1`, and `m2` pointers will be pointing to values that do not belong to them. That's all right as long as we don't try to access them. The next time an input value is filtered, lines 6–8 will reset the pointers to the correct positions.

One advantage of the algorithm we have just developed is that it can be used effectively for either real-time or nonreal-time applications. The value `x` can be taken from either an input port of a DSP system, or from an array of values to be processed (as illustrated). Likewise, the value `y` can be fed to an output port of a DSP system or placed in an output array (as we have done). We will see a program for a complete filtering system in Section 8.4.

8.3 C CODE FOR FIR FILTER IMPLEMENTATION

An FIR filter has no feedback and thus its system diagram can be displayed as shown in Figure 8.4. This configuration represents a convolution involving N coefficients, as described by (8.17).

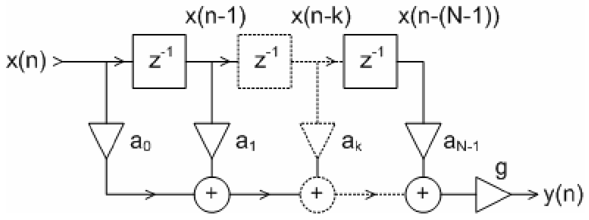


Figure 8.4 System diagram for general FIR filter implementation.

$$y(n) = g \cdot \sum_{k=0}^{N-1} x(n-k) \cdot a(k) \tag{8.17}$$

8.3.1 Real-Time Implementation of FIR Filters

Although the algorithm would appear to be very simple for such an implementation, the fact that we require $N - 1$ values to store the memory states (past values of the input) of the filter complicates the procedure somewhat. We will begin by discussing the real-time implementation problem, and then later we will see that significant time savings can be made with nonreal-time implementations. First, we can generate C code to implement (8.17). We will need to update each memory state after each convolutional sum, which we can do within the same loop if we perform the convolution in reverse order. The simplest way to handle this reverse order is to reverse the coefficients and memory states in their respective arrays, as shown in (8.18) (the coefficient array) and (8.19) (the memory state array). Note that the gain constant is stored as the last entry in the `C` array:

$$C\ \text{array}: \{a_{N-1}, a_{N\text{-}2}, \ldots, a_1, a_0, gain\} \tag{8.18}$$

$$M\ \text{array}: \{x_{\text{-}(N-1)}, x_{\text{-}(N\text{-}2)}, \ldots, x_{\text{-}1}, x_0\} \tag{8.19}$$

The segment of code given in Listing 8.3 shows the basic implementation. We start the process by placing the current value of input `x[0]` into the last position of the memory array `M`. The initial value of the output `o` is calculated using the initial values of the memory state and the coefficient arrays. Then we enter the `for` loop to compute the rest of the convolution sum. Each time through the loop, the memory states are updated so that by the end of the loop all of the memory states occupy the correct position for the next input value. The final step in the process is to multiply the output value by the gain constant of the filter.

```
m[N-1] = x;
o = a[0] * m[0];
for(k = 1; k <= N ;k++)
{ m[k-1] = m[k];
  o += a[k] * m[k];
}
o *= gain;
```

Listing 8.3 Segment of code for FIR filter implementation.

The code of Listing 8.3 will correctly compute the output value but the execution will be slow because of the need for so many index calculations. Listing 8.4 shows the `Dig_FIR_Filt_RT` function for implementing a real-time (RT) digital FIR filter that makes use of pointers to speed the process. In this case, we will need an `M` array of a size equal to the length of the filter (N) and a `C` array that is one larger (N + 1) to accommodate the gain constant. Initially, the `M` array is filled with zeros (unless we know the state of the filter), and the `C` array is filled with the FIR coefficients in reverse order. The final value of the `C` array is the filter's gain constant. We will use movable pointers within the `M` and `C` arrays. The `m1` and `m2` pointers are initially set to the start of the array and then incremented toward the end. By using two movable pointers, we can transfer memory states without any need for address computation. The final step in the process (before converting the output variable back to an integer) is to multiply the accumulated value in `o` by the gain constant of the filter. Although for this code, we obtain the value of input from an array, it could just as well be coming from an input port on a DSP system. Likewise, the output value could be written to an output port instead of an array.

```
/*=====================================================
  Dig_FIR_Filt_RT() - filters input array using FIR
              coefs (uses real-time code)
  Prototype:  void Dig_FIR_Filt_RT(int *X,int *Y,
            double *M,double *C,int numb_coefs,int N);
  Return:     error value.
  Arguments:  X - ptr to input array
              Y - ptr to output array
              M - ptr to memory array
              C - ptr to coefs array
              numb_coefs - number of coefficients
              N - number of values in array
=====================================================*/
void Dig_FIR_Filt_RT(int *X,int *Y,double *M,double *C
                                  ,int numb_coefs,int N)
{ int     *x,*y,    /*  ptrs to in/out arrays */
          i,j;      /*  loop counters */
  double  *c,*m1,*m2, /*  ptrs to coef/memory array */
          o;        /*  output value */

  /*  Make copies of input and output pointers */
  x = X;
  y = Y;
  /*  Start loop for number of data values */
  for(i = 0; i < N ;i++)
  { /*  Make copy of pointers and start loop */
    M[numb_coefs-1] = *x++;
    c = C;
    m1 = m2 = M;
    o = *m1++ * *c++;
    /*  Use convolution method for computation */
    for(j = 1; j < numb_coefs ;j++)
    { *m2++ = *m1;
      o += *m1++ * *c++;
    }
    /*  Multiply by gain, convert to int and store */
    o *= *c;
    *y++ = (int)ceil(o-0.5);
  }
}
```

Listing 8.4 `Dig_FIR_Filt_RT` function.

This implementation of the FIR filter is slow because of the constant memory state shuffle. Note that most sophisticated DSP processors have handled this shuffle by implementing what is referred to as a circular buffer. A circular buffer is one in which the last entry is magically connected to the first entry. In other words, if a pointer that is pointing to the last entry in the buffer is incremented, the processor automatically adjusts the pointer to point to the first entry of the buffer. This helps tremendously, because with a circular buffer, no memory states need to be moved. The newest input value is simply written into the circular buffer over the oldest memory state. The starting point of the convolution is then adjusted to the proper value and the process continues in the normal manner. Although a circular buffer can be simulated on a PC, it requires special coding beyond the scope of this text. In the next section we will discuss a much faster nonreal-time method that we can use instead. We will leave the real-time circular buffer implementation to the DSP chip programmers since DSP systems are usually used for this type of work.

8.3.2 Nonreal-Time Implementation of FIR Filters

The implementation of the FIR filter under nonreal-time conditions can be made much more efficient because we will have available as many of the input values as we would like (and that memory will hold). The input values not only represent the input for the system, but also the memory states of the system. The need for the constant shuffle of past memory states will be removed (to a great degree) as we will see. To understand the efficiency of the nonreal-time process, it may be helpful to view the convolution process graphically. Figure 8.5 shows an array of input values and an array of coefficient values. We can visualize the convolution process as the generation of the sum of products as the coefficients slide past the input values.

In part (a) of Figure 8.5, we see the initial position of the convolution process where x_0 is the first input value to be processed. If previous values of the input are not available then x_{-1} through $x_{-(N-1)}$ would be set to zero. After the convolution sum of products is calculated, the value of y_0 can be determined. The array of coefficients can then be effectively moved one position to the right and the convolution performed again for y_1. This procedure will continue until we reach the end of the input array, processing the last value of the array x_{L-1}, as shown in part (b). Note that since we have a very large array of input values, we can perform many convolution sums without shuffling the past values of input. However, there is usually a limit to how large an input array can be brought into memory. Therefore, we will eventually need to bring in a new array of L input values and start the convolution again from the start of the new array. But in order for the initial convolution sums on this new array to be correct, the values immediately preceding the x_L value must be available at the beginning of the input array, as shown in part (c) of the figure. Consequently, we will need to move the last $N - 1$ values of the initial input array to the beginning on the array. This movement process will have to be accomplished at the end of processing each segment of the input array. If we make the input array buffer much larger than the length of the FIR filter, this amount of processing should be insignificant.

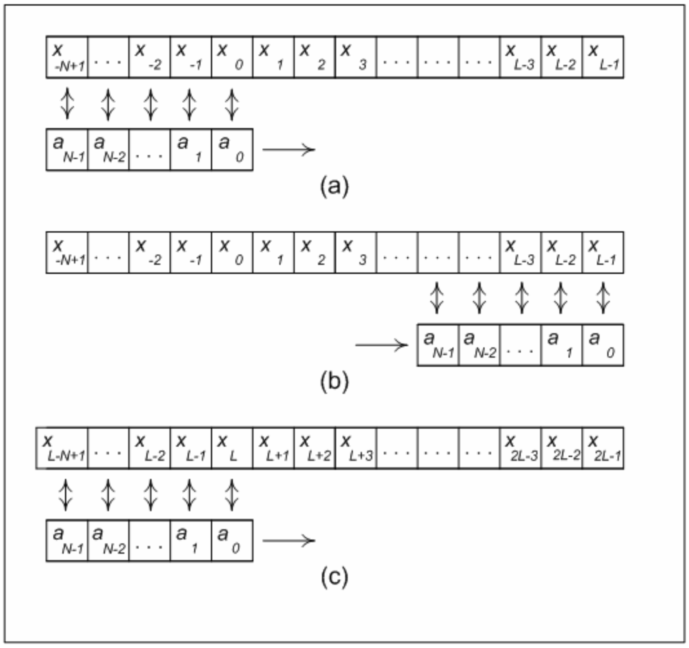


Figure 8.5 Graphical interpretation of convolution.

Listing 8.5 shows the `Dig_FIR_Filt_NRT` function for implementing the nonreal-time (NRT) form of the FIR filter. In this function, the pointers to the input, output, and coefficient arrays are passed as arguments as well as the number of coefficients and number of values to be processed in the input array. Notice that a separate array of memory states is not necessary since the input array also represents the memory states. The function starts by assigning the address of the first input value to the pointer variable *x*, which will march through the array keeping track of the starting point of the convolution. A temporary pointer to the proper output array value is also initialized. The pointer to the coefficient array `c` is initialized within the first `for` loop, which controls the processing of all input values. The convolution sum is then calculated moving the `x` and `c` pointers progressively through the respective arrays accumulating products. After the summation is complete, the next step is to multiply the accumulation by the filter's gain constant, which has been stored at the last entry in the C array. The process is completed by converting the output floating-point value to a fixed-point value and resetting the starting point in the `x` array to the correct position for the calculation of the next output value. After all values in the input array have been processed, the last $N - 1$ values in the array are copied to the beginning of the X array for the processing of the next input segment. As we will see in the next section, when input values are stored in the X array, they are not placed at the beginning of the array (which would overwrite the memory states of the filter), but rather placed at an advanced position in the array based on the length of the filter.

```
/*=====================================================
  Dig_FIR_Filt_NRT() - filters input array using FIR
              coefs (uses nonreal-time code)
  Prototype:  void Dig_FIR_Filt_NRT(int *X,int *Y,
                      double *C,int numb_coefs,int N);
  Return:     error value.
  Arguments:  X - ptr to input array
              Y - ptr to output array
              C - ptr to coefs array
              numb_coefs - number of coefficients
              N - number of values in array
=====================================================*/
void Dig_FIR_Filt_NRT(int *X,int *Y,double *C,
                                 int numb_coefs,int N)
{ int     *x,*y,    /*  ptrs to in/out arrays */
          i,j;      /*  loop counters */
  double  *c,       /*  ptrs to coefficient array */
          o;        /*  output value */

  /*  Make copies of input and output pointers */
  x = X;
  y = Y;
  /*  Start loop for number of data values */
  for(i = 0; i < N ;i++)
  { /*  Make copy of coefs pointer and start loop */
    c = C;
    o = *x++ * *c++;
    /*  Use convolution method for computation */
    for(j = 1; j < numb_coefs ;j++)
    { o += *x++ * *c++;}
    /*  Multiply by gain, convert to int and store */
    o *= *c;
    *y++ = (int)ceil(o-0.5);
    /*  Reset the pointer in input data */
    x -= (numb_coefs - 1);
  }
  /*  Copy last values to front of buffer */
  memcpy(X,&X[CHUNK_SIZE],2*(numb_coefs-1));
}
```

Listing 8.5 `Dig_FIR_Filt_NRT` function.

8.4 FILTERING SOUND FILES

We have now developed several ways to implement the digital filters that we designed in the previous chapters. It is now time to actually implement them and listen to the results. Dedicated DSP processors may not be available to us, but many of us do have sound cards in our computers, so we can at least implement the nonreal-time versions of the filters. The manner in which this will be accomplished is the following. Two sound files have been included on the software disc included with this text, and we can record other sound files using our sound cards. These sound files can be filtered using WFilter, which has been included with this text. After filtering the sound files, we can compare the results using the sound card to play the original and filtered versions of the sound files. (Please read the documentation for your sound card to determine how to record and play sound files.) To get further information on sound file formats and the C code used in filtering the files, please refer to Appendix I.

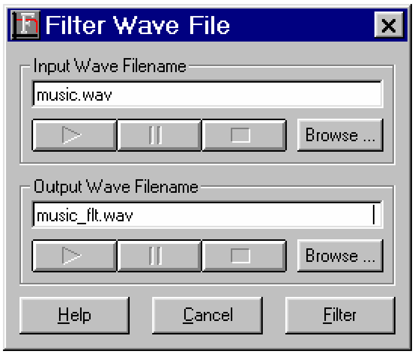
To demonstrate the ease of filtering waveforms, we'll design a digital filter and use it to filter one of the waveforms on the software disc in our next example.

Example 8.1 IIR Digital Filtering of Waveform

Problem: Determine the effects on music.wav (available on the CD) when it is filtered by a digital Butterworth IIR filter with the following characteristics:

a_{pass} = –0.5 dB, a_{stop} = –60 dB, f_{pass} = 800 Hz, and f_{stop} = 1,600 Hz

Solution: First, we design the filter using WFilter. Since we will be filtering a file sampled at 22,050 Hz, we use that value as the sampling frequency. After designing the filter, we can select ***Filter Wave File*** from the ***Options*** menu. The following dialog box will be shown.


Filter Wave File

Input Wave Filename: music.wav

Output Wave Filename: music_flt.wav

Figure 8.6 Filter Wave File dialog box.

We can identify the file to be filtered by typing directly into the dialog box or by using the browse button to locate the file. After specifying the input and output file names, simply press the ***Filter*** button and the filtering action will take place. You will be able to tell when the filtering has finished by the play buttons being enabled. Now we can compare the two sound files by playing each file individually. Notice that the lowpass filter was successful in eliminating the chimes from the original version.

Other filters can be created and other sound files can be filtered. If you use the speech.wav file on the accompanying disc be aware that it has a sampling frequency of 11,025 Hz. It is also captured as an 8-bit file.

8.5 CONCLUSION

We have reached the end of this chapter where the implementation of FIR and IIR filters has been developed. We have developed a fully functional program to filter sound files, which can be mono or stereo, and 8-bit or 16-bit files. Other file formats can be added to the program with minimum effort by writing functions to read and write the file headers and passing the required information to our functions. Compressed files can be handled by implementing a decompression function between the reading of the data and its filtering. Further information on sound file formats and C code for filtering can be found in Appendix I.

Chapter 9

Digital Filtering Using the FFT

At this point we have discussed the design and implementation of digital filters. In the process we have investigated the characteristics of the input and output signals in both the time and frequency domains. It is time now to investigate a more direct relationship between the time domain and frequency domain for discrete time systems. We will begin by discussing the discrete time version of the Fourier transform. After the discrete Fourier transform (DFT) discussion, we will learn about the more computational efficient version called the fast Fourier transform (FFT). The C code for the FFT will be developed, and finally, we will take a look at one method of using the FFT in linear filtering.

9.1 THE DISCRETE FOURIER TRANSFORM (DFT)

The Discrete Fourier Transform (DFT) can be used to compute the frequency content of any discrete time signal. Consider first (9.1), where ω is periodic with a period of 2π. (Remember that a radian frequency of 2π in the z-plane is equivalent to the sampling frequency (F_s) in the s-plane.) In (9.1), $x(n)$ represents the time domain signal, which has an infinite number of samples. The spectrum that will result from sampling an analog signal will actually be many replicas of the analog spectrum spaced at multiples of the sampling frequency, as shown in Figure 5.2 in Chapter 5. We will be able to select just one of these spectrums by using a filter at the output of our discrete time system.

$$X(\omega) = \sum_{n=-\infty}^{\infty} x(n) \cdot e^{-j\omega n} \tag{9.1}$$

We see that although (9.1) correctly defines the Fourier transform for a discrete time signal, it is impossible to implement for two reasons. First, we will never be able to obtain *all* of the time domain samples, and second, we will never be able to evaluate the equation at *all* values of the frequency variable ω.

Therefore, the first adjustment we make to our strict definition is to modify (9.1) to reflect the fact that we will have only a finite number of samples of *x(n)* with which to work. Equation (9.2) shows the definition when we assume that we have only *N* samples of the signal data:

$$X(\omega) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j\omega n} \tag{9.2}$$

Truncating the signal sequence as in (9.2) is effectively applying a rectangular window to the time domain sequence. This causes problems in the resulting frequency domain description which can be lessened by applying a different window such as one used in Chapter 7 dealing with FIR filter coefficients. The impact of such a window will be discussed very soon.

If we further assume that we'll only need to evaluate the frequency response data at a finite number of equally spaced frequencies from 0 – 2π, we have the definition of the *K*-point DFT of an *N*-point signal.

$$X(\omega_k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j\omega_k n},\ k = 0, 1, \ldots, K\text{-}1 \qquad \text{where } \omega_k = \frac{2\pi k}{K} \tag{9.3}$$

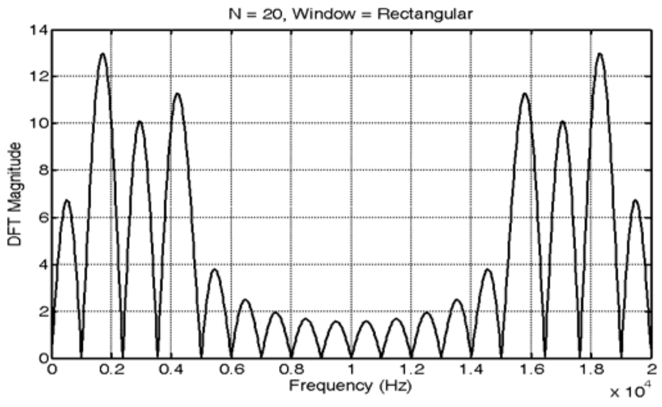
<u>Example 9.1 Calculation of DFT with Rectangular Window</u>

Problem: Determine the DFT of an audio signal containing three distinct frequencies of F_1 = 2 kHz, F_2 = 3 kHz, and F_3 = 4 kHz. Assume that the sampling frequency is F_s = 20 kHz and that we make use of either 20 or 40 samples of the waveform. Use a rectangular window (in other words, simply truncate the sequence).

Solution: First, we generate a waveform containing the three frequencies.

$$x(n) = \sin(2\pi F_1 n / F_s) + \sin(2\pi F_2 n / F_s) + \sin(2\pi F_3 n / F_s)$$

Then, we calculate the DFT using (9.3) letting *K* = 1,000 points. (This gives us a near continuous frequency response.) The results are shown in Figure 9.1. Although not shown, if we had used a large number of data points (*N* = 1,000), the DFT graphs would have shown six very distinct spikes in the frequency domain with far less "clutter" at other frequencies. (There are six major responses in the spectrum because the content from 10 kHz to 20 kHz is a mirror reflection of the content from 0 Hz to 10 kHz. We need only to concern ourselves with the first half of the spectrum.) As it is, we can see that the three major spikes in the first half of the spectrum generally reflect the three frequencies of our waveform. As *N* increases, the position will become more and more precise. However, we should be concerned about the clutter at other frequencies and why it is reduced as *N* increases.



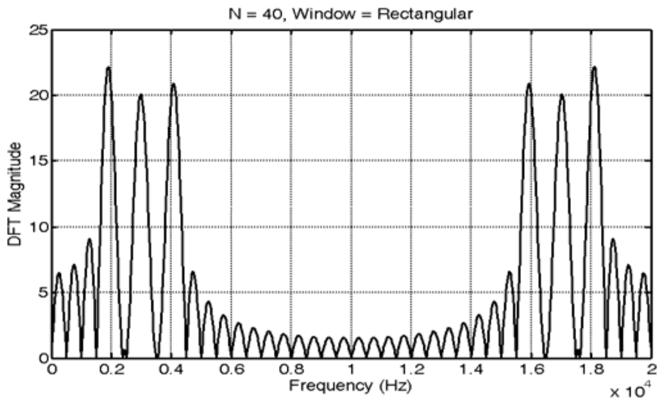


Figure 9.1 DFT of three frequencies with rectangular window.

The explanation of the numerous extraneous lobes contained in the DFT responses goes back to the time domain and our truncation of $x(n)$. By truncating the sequence, we effectively multiplied two time domain functions together, as shown in (9.4). (The rectangular window function $w(n)$ in our example will have 20 or 40 values of one and zeros for all other values.)

$$x_{trun}(n) = x(n) \cdot w(n) \tag{9.4}$$

When two waveforms are multiplied in the time domain, their frequency spectrums are convolved in the frequency domain. Although the frequency spectrum of the signal will be three spikes, the frequency spectrum of the rectangular window will be a *sinc* function shown in Figure 9.2. Notice all of the side lobes produced. When convolved with the three spikes representing the three signal frequencies, these side lobes will create the extraneous side lobes shown in Figure 9.1. By contrast, when viewing the response of the transform of the Hamming window, we see no annoying side lobes, but a smaller and wider main lobe. The results of using a Hamming window is shown in Example 9.2.

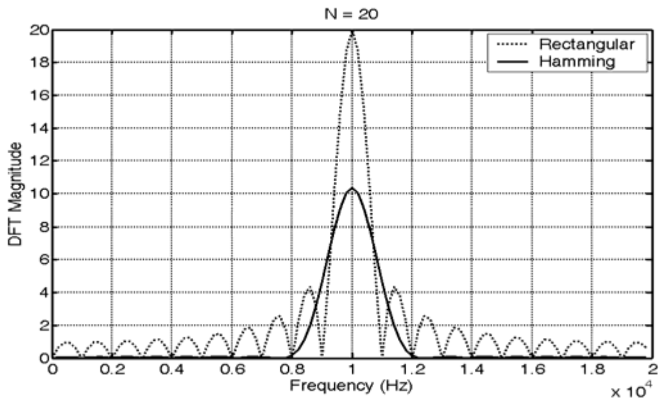


Figure 9.2 DFT of rectangular and Hamming windows.

<u>Example 9.2 Calculation of DFT with Hamming Window</u>

Problem: Repeat Example 9.1, but use a Hamming window on the truncated input sequence.

Solution: The input sequence $x(n)$ is formed the same way as in Example 9.1, but the window function $w(n)$ takes on the following values:

$$w(n) = 0.54 - 0.46 \cdot \cos(2\pi n/(N-1))$$

After calculating the DFT of the product $x(n)\cdot w(n)$, the results are displayed in Figure 9.3 with a number of interesting comparisons to Figure 9.1. First, we notice that the clutter of the side lobes of Figure 9.1 has been reduced to a great degree. However, we also notice that for the 20-point signal case, there are no longer three, nearly equal spikes in the first half of the spectrum. Even in the 40-point case, the three spikes are no longer as clearly distinct. These results derive from the fact that the transform of the Hamming window is approximately twice as wide as the main lobe of the rectangular window transform. The resulting resolution of the transform is therefore not as crisp as in the rectangular case.

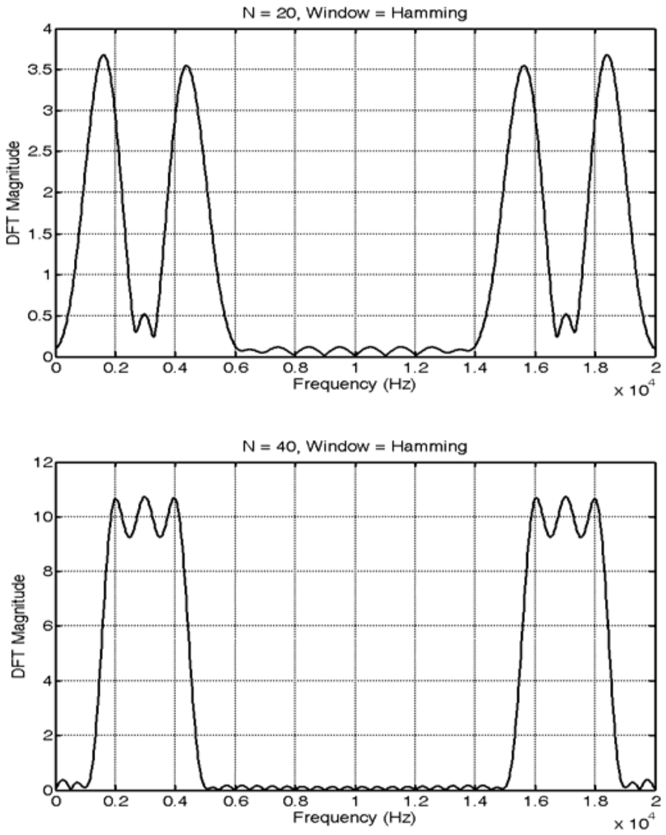


Figure 9.3 DFT of three frequency lengths with Hamming window.

It can be shown that the frequency resolution Δf of the DFT is improved by increasing the number of time domain samples, as shown in (9.5). As N increases, the ability to see smaller frequency details improves regardless of window type used. The coefficient m_w is a windowing factor that takes on a value of 1 for a rectangular window, approximately 2 for a Hamming window, and other values for Kaiser based on the β value chosen. (See Orfanidis' text for further details.)

$$\Delta f = m_w \cdot \frac{f_s}{N} = m_w \cdot \frac{1}{T_s \cdot N} \tag{9.5}$$

Example 9.3 Determine Resolution for DFT System

Problem: Determine the number of time domain samples needed to resolve the three frequencies of Examples 9.1 and 9.2.

Solution: In each case, the resolution Δf required is 1 kHz in order to clearly see the three frequencies. In Example 9.1, m_w will be 1, and for Example 9.2, we will use a value of 2. Equation (9.3) can then be used to determine that the number of time samples N needed would be 20 for Example 9.1 and 40 for Example 9.2. Figures 9.1 and 9.3 seem to support these values.

From the previous examples we see that we must choose N carefully to get the resolution required for our system. It seems obvious that the more time domain samples available, the clearer the picture of the frequency spectrum will be, and that is true. But once the minimum number of data samples is determined, is there any advantage to increasing the number of frequency points computed? It may not be clear at this point, but providing more frequency points does *not* improve the resolution, it simply provides a more continuous graph of the spectrum. This is seen in Figure 9.4, which shows the Hamming window case with $N = 20$ and $K = 20$. Comparing this to the 20-point graph in Figure 9.3, we note that the 3-kHz component is obscured in either case. Increasing the number of frequency data points 50-fold did nothing to improve the resolution of the DFT.

In a similar argument, if you have available many time samples, but compute far fewer frequency points, you are losing valuable frequency information. Therefore, a general rule of thumb is to let $N = K$ as a starting point in the design of a DFT system. On occasion, it may be necessary to allow K to increase to find a more exact value of the frequency of a maximum.

9.2 THE FAST FOURIER TRANSFORM (FFT)

While the computation of the DFT may be straightforward, it is not without computational cost. For an N-point DFT, each frequency point will require N multiplications of the real valued $x(n)$ times the complex valued $exp(-j\omega_k n)$. (This operation effectively requires two real multipliers.) Then, assuming there are N different frequency points, there will be N^2 complex multiplications necessary for the computation of the DFT. This can produce large numbers very quickly.

The fast Fourier transform (FFT) had its beginnings in 1965, long before the DSP chip was even a dream of engineers. However, it took the computational power of computers to bring its importance to the attention of designers. It is sometimes thought that the FFT produces a different result than the DFT, but it does not. The FFT is simply a faster method of computing the DFT. Derivations of the FFT algorithm can be found in the references in Appendix A, but it is worthwhile to see a simple illustration of how computational savings can be made using the FFT.

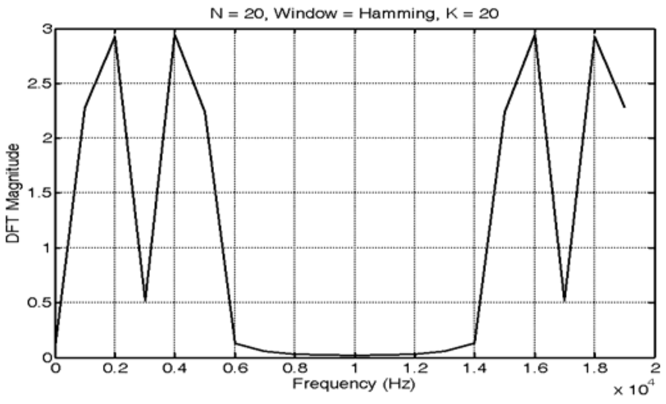


Figure 9.4 DFT with Hamming window and K = 20.

9.2.1 The Derivation of the FFT

To begin, consider (9.6), which is a streamlined version of (9.2). In it we replace ω_k with the index k, and let W_N replace the cumbersome $exp(-j2\pi/N)$. (These W_N^{kn} terms are often referred to as "twiddle factors.")

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{kn}, k = 0, 1, \ldots, N\text{-}1 \tag{9.6}$$

We can now look at a simple 4-point DFT example. If we expand (9.6) into four separate equations representing the four frequency points, we have

$$\begin{aligned}
X(0) &= x(0) \cdot W_4^0 + x(1) \cdot W_4^0 + x(2) \cdot W_4^0 + x(3) \cdot W_4^0 \\
X(1) &= x(0) \cdot W_4^0 + x(1) \cdot W_4^1 + x(2) \cdot W_4^2 + x(3) \cdot W_4^3 \\
X(2) &= x(0) \cdot W_4^0 + x(1) \cdot W_4^2 + x(2) \cdot W_4^4 + x(3) \cdot W_4^6 \\
X(3) &= x(0) \cdot W_4^0 + x(1) \cdot W_4^3 + x(2) \cdot W_4^6 + x(3) \cdot W_4^9
\end{aligned} \tag{9.7}$$

Using the values of the twiddle factors in this special case we can represent this process as a matrix multiplication as follows:

$$\mathbf{X} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & 1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \cdot \mathbf{x} \tag{9.8}$$

Another popular method describing the mathematical process, and one that better describes the computational efficiency we are looking for, is to use signal flow graphs, as shown in Figure 9.5. In this description, all branches of the graph have a weight (or multiplier) of 1 except as noted.

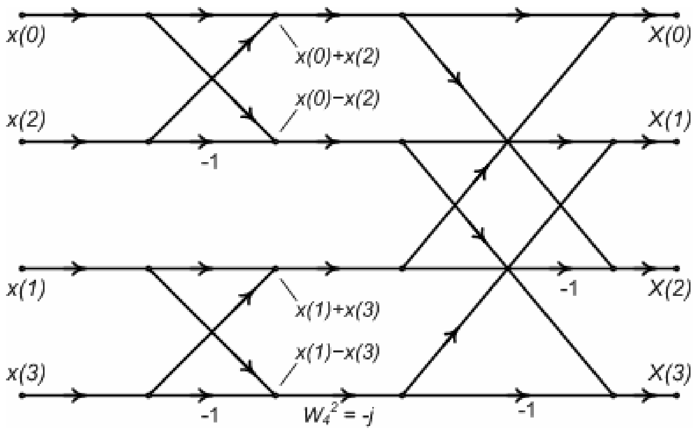


Figure 9.5 Four point decimation-in-time FFT.

It is clear from this illustration that few multiplications are necessary and also that samples are combined at intermediate points along the path. For example, in the middle of the diagram we see that combinations of x_0 and x_2 as well as x_1 and x_3 are made. These combinations are then processed as a unit that saves individual computations. So instead of x_0 being involved in four multiplications as shown in (9.7), it is involved only in two. This efficiency is continued for larger FFTs. For example, if an 8-point DFT were required, two 4-point DFTs would be used with an additional butterfly stage. (The distinctive cross-linked structure in Figure 9.5 is called a butterfly because if you rotate it 90° that is what it looks like.) As the length of the DFT increases, the number of butterfly stages increases as well, which produces the efficiency. The net effect is that the FFT algorithm would require $N{\cdot}log_2(N)$ complex multiplications instead of N^2 as in the DFT case. The resulting efficiency can be seen in Table 9.1.

One interesting feature of the process shown in Figure 9.5 is that the frequency points along the right side of the graph are in sequence, while the time points are in shuffled order. This particular ordering is described as the decimation-in-time (DIT) transform. There is also a decimation-in-frequency (DIF) algorithm in which the time-domain samples remain in sequence, but the frequency components are shuffled. Although the shuffled sequence may look arbitrary, we will see that the ordering can be exploited with a novel addressing mode available in most DSP processors. As an example, consider an 8-point FFT looking only at the input sequence as shown in Table 9.2. Column 1 shows the shuffled order of *x(n)* as being *x*(*0*), *x*(*4*), *x*(*2*), and so forth. Column 2 shows the binary equivalent of the index. Column 3 shows the result of reversing the bits of column 2. Finally, the last column shows that the indices are in order if viewed as bit-reversed values. This feature is exploited in DSP chips and is a common addressing mode used for FFTs, one of the many efficiencies built into DSP chips.

Table 9.1

Comparison of DFT and FFT Computations

N	*DFT*	*FFT*	*Ratio (DFT/FFT)*
16	256	64	4
64	4,096	384	10.7
256	65,536	2,048	32
1,024	1.05E06	10,240	102
4,096	16.8E06	49,152	341
16,384	268.E06	229,380	1,170
65,536	4.29E09	1.05E06	4,096

Table 9.2

Bit-Reversed Addressing

DIT order	*Index as binary*	*Bit-reversed index*	*Bit-reversed order*
0	000	000	0
4	100	001	1
2	010	010	2
6	110	011	3
1	001	100	4
5	101	101	5
3	011	110	6
7	111	111	7

9.2.2 The Inverse FFT

Once the FFT has been computed, we have information about the frequency content of the signal. There are many cases when we will need to process this frequency information in some way and then want to transform the frequency content back to the time domain. For that we will need the inverse FFT. Because of the symmetry of the transform, it can be shown that the inverse FFT can be based on the FFT with simple operations both before and after the FFT, as shown in (9.9). As indicated, the inverse FFT of a sequence can be found by finding the FFT of the conjugate of the sequence, and then conjugating the result and dividing by *N* (the length of the FFT). We will make use of this fact in Section 9.4. At this point we are ready to discuss the C code necessary to implement the FFT algorithm.

$$FFT^{-1}(X) = \frac{[FFT(X^{*})]^{*}}{N} \tag{9.9}$$

9.3 C CODE FOR THE FFT

As we begin the discussion of the C code for computation of the FFT, it is important to note the differences between programming for a general purpose processor and a DSP chip. The DSP chip is a microprocessor designed to implement the types of computations that are commonplace in digital signal processing applications. These include complex arithmetic, convolution, and fast Fourier transforms. DSP processors have multiple data and control buses and highly efficient instructional commands. For example, most DSP processors will be able to accomplish a multiply-add instruction in only one instruction cycle while general purpose processors would require many more.

Another major difference is the fact that DSP chips have an addressing mode that includes bit-reversed addressing. This eliminates the shuffling of data points (either time-domain or frequency-domain) that we need in our general purpose code. As well, DSP processors have dual memories to hold the real and imaginary parts of complex numbers, while we have to implement a complex data type to handle the FFT computations. Therefore, it is recommended that if your project will be implemented on a DSP chip, you use the features of the software for that chip to most efficiently implement your project. But if you have a need to implement the FFT on a general purpose processor, we will develop that code now.

There are a number of FFT algorithms discussed in the available references, but we will discuss the common decimation-in-time radix-2 algorithm. A radix-2 implementation requires that the number of input data points be equal to a power of two, as shown in (9.10), where B is an integer. Although this may first appear to be a limitation, it is not. Any input sequence can simply be increased to the required size by adding zeros to the end of the sequence. This operation is referred to as “padding” the sequence and in no way affects the computation of the FFT.

$$N = 2^B \tag{9.10}$$

Listing 9.1 shows `FastFT`, a function to compute the DIT implementation of the FFT. This function has been adapted from Orfanidis’ text listed in Appendix A and is similar to others described in the literature. The input signal has been placed into a complex array `X` since this particular algorithm will perform an “in-place” computation of the FFT. That means that after the FFT has been completed, the resulting complex valued FFT will reside in the array `X`. This will make the use of memory more efficient. In addition to the input array, the function also receives the length of the FFT and the number of bits required to describe each index.

The first step in the procedure is to swap the values in the input array as required in a DIT implementation. This is accomplished in the first loop and makes use of the `RevBits` function shown in Listing 9.2 (also adapted from the Orfanidis text). After the input values have been swapped, the FFT is computed by means of sequential calculation of the butterfly stages with the required twiddle factors computed first.

```
/*==============================================================
  FastFT() - radix-2 decimation-in-time Fast Fourier Transform
  Prototype:  void FastFT(short Length, short Bits, complex *X);
  Arguments:  Length - length of FFT
              Bits - Length = 2^Bits
              X - ptr to array of complex numbers
  Return:     none (conversion done in place)
==============================================================*/
void FastFT(short Length, short Bits, complex *X)
{ long     k, i, p, q,       /*  counters and indices */
           M;                /*  stage of FFT */
  complex  A, B, V, W, Tmp;  /*  complex constants */
  short    n,                /* counter */
           RIndex;           /*  reversed ndex */

  /* Swap the values in x */
  for(n = 0; n < Length; n++)
  { RIndex = RevBits(n,Bits);     /*  get reversed index */
    if (RIndex < n) continue;     /*  only need to swap half */
    Tmp = *(X+n); *(X+n) = *(X+RIndex); *(X+RIndex) = Tmp;
  }

  /* Implement the butterfly in stages */
  M = 2;
  while (M <= Length)  /* while loop controls stages of FFT */
  { W.re = cos(2*PI / M);         /* twiddle factors */
    W.im = sin(2*PI / M);
    V.re = 1;
    V.im = 0;
    for(k = 0; k < M/2; k++)      /* index through stages */
    { for(i = 0; i < Length; i += M)
      { p = k + i;
        q = p + M / 2;
        A = X[p];
        B = cmul(X[q],V);
        X[p] = cadd(A,B);         /* butterfly operations */
        X[q] = csub(A,B);
      }
      V = cmul(V,W);
    }
    M = 2 * M;
  }
}
```

Listing 9.1 `FastFT` function.

These two functions can now be used to compute the FFT of any input sequence. All that is required is that the sequence be adjusted to a length that is a power of two by padding the end of the sequence with zeros. The function can then be provided with the pointer to the sequence, the length of the sequence, and the number of bits in the indices. After the function has completed its work, the array `X` will hold the complex valued FFT. These values will be stored as the real and imaginary values of the transform, but can be better interpreted as the magnitude and phase angle of the transform. The conversion from one form to another is shown in (9.11)–(9.12). When computing the phase angle you should use the `atan2` function that takes both the real and imaginary values as arguments. This guarantees that the angle will be placed in the correct quadrant.

```
/*============================================================
  RevBits() - reverses the bits in an integer
  Prototype:  short RevBits(short Input, short NumBits);
  Arguments:  Input - integer to reverse
              NumBits - number of bits used
  Return:     integer with bits reversed
============================================================*/
/* Adapted from Orfanidis - Intro. to Signal Processing */
short RevBits(short Input, short NumBits)
{   short i,        /*  loop counter */
          RevInput; /*  interger with reversed bit values */
  RevInput = 0;
  /*       Loop through Input, bit by bit. If bit is set, set
           appropriate bit in RevInput, and clear bit in Input.*/
  for(i = NumBits-1; i >= 0 ;i--)
  { if ((Input >> i) == 1)
    { RevInput += (1 << (NumBits-1-i));
      Input -= (1 << i);
    }
  }
  return RevInput;
}
```

Listing 9.2 `RevBits` function.

$$Magnitude = \sqrt{\mathrm{Re}(X)^2 + \mathrm{Im}(X)^2} \tag{9.11}$$

$$Angle = \arctan(\mathrm{Im}(X)/\mathrm{Re}(X)) \tag{9.12}$$

The interpretation of the resulting frequency response is easy as long as we remember that the values of the FFT will be spread from zero frequency up to the sampling frequency. Therefore, if we had a 1,024-point FFT computed on a sequence that had been sampled at a frequency of 20 kHz, the values in the transformed sequence would be spaced every 19.53 Hz (20 kHz ÷ 1024). This would mean that the first sample was the response at 0 Hz (or the DC value), and the last value would represent the response at 19,980.47 Hz. Remember that the last half of the response will be a mirror image of the first half. Therefore, for this particular case, only the first half of the FFT values would need to be analyzed. The others will simply be duplicates.

9.4 APPLICATION OF FFT TO FILTERING

There are many applications for the FFT. Frequency analysis, spectral densities, and filtering are among the more popular. Since this is a text about filtering, we discuss how the FFT can be used in filtering applications. To begin this discussion, we review the process of FIR filtering first. Once FIR filter coefficients have been determined, the output signal of the filter can be determined from the input signal by convolving the filter coefficients and the input signal, as shown in (9.13). We learned in Chapter 5 that convolution in the time domain was equivalent to simple multiplication in the frequency domain, as shown in (9.14).

$$y(n) = h(n) * x(n) = \sum_{k=0}^{M-1} h(k) \cdot x(n-k), n = 0, 1, \ldots, N-1 \tag{9.13}$$

$$Y(z) = H(z) \cdot X(z) \tag{9.14}$$

Since we now have a fast algorithm for determining the transform of a time-domain signal, it may be useful to consider the alternative presented by (9.14). Let's consider the steps involved in using transforms to implement the filtering process:

1. Transform the filter coeffients: $H(z) = FFT\, [\, h(n)\,]$.
2. Transform the input signal: $X(z) = FFT\, [\, x(n)\,]$.
3. Multiply the two complex sequences: $Y(z) = H(z){\cdot}X(z)$.
4. Inverse transform to find the output: $y(n) = FFT^{-1}\, [\, Y(z)\,]$.

The process outlined above may or may not prove to be faster than the convolution of (9.13). It all depends on the number of filter coefficients and the number of points in the FFT. The text by Proakis and Manolakis mentioned in Appendix A offers a detailed comparison that will be discussed further at the end of this section. Listing 9.3 shows `Dig_FFT_Filt` that performs steps 2 through 4 above. Note that it is assumed that the filter coefficients have already been transformed and stored in *H*. The function first transforms the input signal *X*, then multiplies the coefficients times the transform of *X*, and finally computes the inverse FFT. Note that the results of each of these steps are stored in *X*.

There are many cases when the length of the input signal exceeds the size of the FFT that can be applied. In that case, the signal must be broken into smaller, more manageable, sections and the FFT filtering algorithm applied to each section. Unfortunately, it is not as simple as taking each *N*-point group of input samples and generating an output group. The resulting patched-together sequence would undoubtedly have discontinuities at the combination points. This problem can be alleviated by keeping some processed samples from each grouping and using them as the initial points in the next sequence. This is exactly what was done in the FIR filtering code described in Section 8.3.2 and represents providing initial conditions to the filtering process. When applied in FFT filtering this process is often called the overlap-and-save method.

```
/*============================================================
  Dig_FFT_Filt() - performs FFT filtering
  Prototype:  void Dig_FFT_Filt(complex *X, complex *H,
           short numb_coefs, short FFT_size, short Bits);
  Return:     error value.
  Arguments:  X - input data
              H - transform of filter coefs
              numb_coefs - number of filter coefs
              FFT_size - number of pts in FFT
              Bits - FFT_size = 2^Bits
============================================================*/
void Dig_FFT_Filt(complex *X, complex *H,
                                  short FFT_size, short Bits)
{   short i;
    /* Perform FFT on X */
    FastFFT(FFT_size,Bits,X);

    /* Multiply X and H */
    for (i = 0; i < FFT_size; i++)
    { X[i] = cmul(X[i],H[i]);}

    /* Get inverse FFT of X */
    /* Conjugate input */
    for (i = 0; i < FFT_size; i++)
    { X[i] = cconj(X[i]);}

    /* Perform FFT */
    FastFFT(FFT_size,Bits,X);

    /* Conjugate again */
    for (i = 0; i < FFT_size; i++)
    { X[i] = cconj(X[i]);}

    /* X now holds the filtered data */
    return;
}
```

Listing 9.3 `Dig_FFT_Filt` function.

Figure 9.6 shows the process. The input signal is padded with *M*–1 initial zeros (assuming there are *M* filter coefficients). Then, the remaining signal is broken into sections containing *L* values where *L* is defined in (9.15). As pictured in the diagram, there will be *N* samples transformed in each operation, but the first *M*–1 will be the values saved from the last *M*–1 values of the previous operation (except for the initial grouping, which will have *M*–1 zeros.) Once the FFT filtering has been completed on the grouping, the last *L* values of the sequence are saved, discarding the first *M*–1 values. This operation will result in an error-free result when the *y* groupings are reassembled.

$$L = N - (M - 1) \tag{9.15}$$

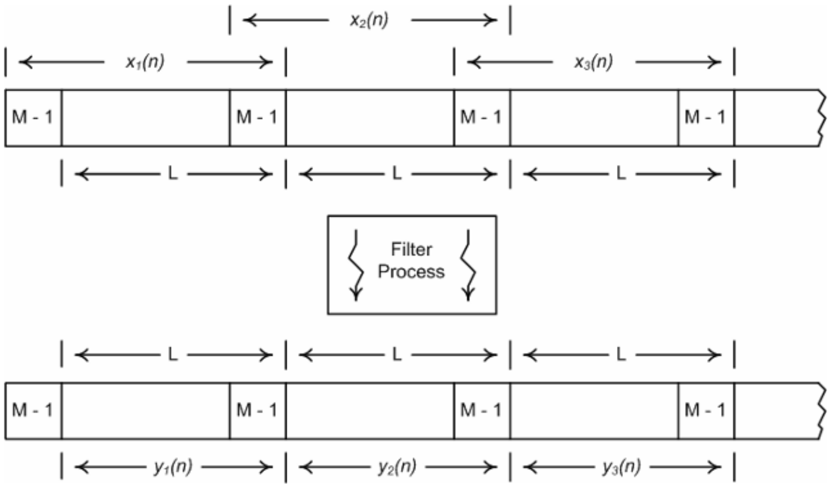
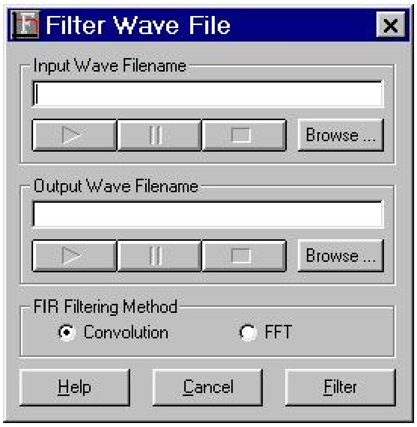


Figure 9.6 Using the overlap-and-save method for linear FFT filtering.

WFilter does have an option for FFT filtering available if an FIR filter has been designed. Once the filter has been designed, the user can elect to filter a WAV file by selecting *Options->Filter Wave File.* The *Filter Wave File* dialog box will appear, as shown in Figure 9.7, and will provide the user the option of filtering the waveform via the more traditional method of convolution as discussed in Chapter 7, or using the FFT techniques discussed in this chapter.



FIR Filtering Method
[✓ Convolution] FFT

Figure 9.7 Filter Wave File dialog box.

Unfortunately, the true speed of the FFT technique cannot be realized on a general-purpose processor that many users will be using to filter waveforms. As mentioned previously, the general-purpose processor will not have the option of bit-reversed addressing or manipulation of complex numbers in two memory locations. Therefore, the true efficiency of the FFT algorithm cannot be demonstrated on a general-purpose processor.

However, Proakis and Manolakis have discussed the theoretical efficiencies of this process that we can review at this point. They have determined that the number of complex multiplications required per output point can be determined by (9.16). Recognizing that there are four real multiplications needed for each complex multiplication, and also recognizing that for convolution there are *M* real multiplications needed for each output point, a comparison can be made in Table 9.3. In the table, the left-hand column represents the number of data points *N* in the FFT. The top row gives the various numbers of filter coefficients used *M*. Within the table are given the number of real multiplications per output point necessary for the calculation of the FFT. These numbers should be compared to the *M* value, which represents the number of calculations per output point for the convolution method. Not all filter applications should consider the FFT approach as depicted in column 2 (*M* = 32). In the best case, the FFT approach would require 41 real multiplications to the convolution approach of 32. However, it is apparent that as the number of filter coefficients increase, the FFT approach can provide real efficiencies. A second point to recognize is that the number of points in the FFT should also be considered when setting up the filtering system. In general, it appears that the size of the FFT should be set to approximately 8 times the number of coefficients used in the filter.

$$c = \frac{N \cdot \log_2(2 \cdot N)}{L} \tag{9.16}$$

Table 9.3

Number of Real Multiplications per Output Point

N=FFT size	*M=32*	*M=64*	*M=128*	*M=256*	*M=512*
64	54	-	-	-	-
128	43	64	-	-	-
256	41	48	72	-	-
512	43	46	54	80	-
1,024	46	47	51	59	88
2,048	49	50	52	55	64
4,096	53	53	54	56	60
8,192	57	56	57	58	60

9.5 CONCLUSION

This completes our chapter on the FFT and its application to linear filtering. The FFT is a valuable function that provides a link from the time domain to the frequency domain. We have found that FFT will find application to linear filtering when the number of coefficients is large enough to warrant the more complex computations.

This is also the conclusion for the text. Along the way, we have developed some of the most time-honored methods used in analog and digital filter design. By no means have we explored every nuance of filter design, and we see new techniques being developed every day. However, we have developed the key approximation techniques used today. The methods used can serve as a template in the development of other approximations. We have been able to adjust normalized functions for use in a variety of selectivities. We have shown one method of implementing analog filters and discussed some of the potential pitfalls that must be considered.

After the introduction of some principles of discrete-time theory, we learned how to design both FIR and IIR digital filters using two entirely different techniques. C code for the implementation of these filter types was developed and analyzed. And finally, we finished with the discussion of the FFT in filtering applications. I hope this text and the accompanying software disc will serve as a starting point for your journey into the exciting field of filter design.

Appendix A

Technical References

This appendix includes a list of references appropriate for the study of analog and digital filter design methods using C.

ADVANCED MATHEMATICS REFERENCES

Abramowitz, Milton, and Stegun, Irene, eds., *Handbook of Mathematical Functions*, Dover Publications, Inc., New York, 1965.

Morris, John L., *Computational Methods in Elementary Numerical Analysis,* John Wiley & Sons, New York, 1983.

Rice, John R., *Numerical Methods, Software, and Analysis,* 2nd ed., Academic Press, Inc., San Diego, CA, 1993.

Spiegel, Murray R., *Mathematical Handbook of Formulas and Tables*, Schaum's Outline in Mathematics, McGraw-Hill Book Co., New York, 1968.

ANALOG FILTER DESIGN REFERENCES

Daniels, Richard W., *Approximation Methods for Electronic Filter Design*, McGraw-Hill Book Company, New York, 1974.

Daryanani, Gobind, *Principles of Active Network Synthesis and Design*, John Wiley & Sons, New York, 1976.

Johnson, D. E., Johnson, J. R., and Moore, H. P., *A Handbook of Active Filters*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1980.

Moschytz, G. S., and Horn, P., *Active Filter Design Handbook,* John Wiley & Sons, New York, 1981.

Sedra, Adel S., and Brackett, Peter O., *Filter Theory and Design: Active and Passive*, Matrix Publishers, Inc., Champaign, IL, 1978.

Schaumann, R., Ghausi, Mohammed S., and Laker, Kenneth R., *Design of Analog Filters: Passive, Active RC and Switched Capacitor*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1990.

Schaumann, R., and Van Valkenburg, M. E., *Design of Analog Filters*, Oxford University Press, New York, 2001.

C PROGRAMMING REFERENCES

Bronson, Gary, *C for Engineers and Scientists*, West Publishing Co., New York, 1993.

Deitel, H. M., and Deitel, P. J., *C How to Program*, 2nd ed., Prentice Hall, Inc., Englewood Cliffs, NJ, 1994.

Hanly, J. R., Koffman, E., and Friedman, F., *Problem Solving and Program Design in C*, Addison-Wesley Publishing Co., Reading, MA, 1993.

Kernighan, Brian W., and Ritchie, Dennis M., *The C Programming Language*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1978.

Maguire, Steve, *Writing Solid Code*, Microsoft Press, Redmond, WA, 1993.

McConnell, Steve, *Code Complete*, Microsoft Press, Redmond, WA, 1993.

Waite, Mitchell, and Prata, Stephen, *The Waite Group's New C Primer Plus*, Howard W. Sams & Co., Carmel, IN, 1990.

DIGITAL FILTER DESIGN REFERENCES

Antoniou, Andreas, *Digital Filters: Analysis, Design and Applications*, 2nd ed., McGraw-Hill, Inc., New York, 1993.

Embree, Paul M., and Kimble, Bruce, *C Language Algorithms for Digital Signal Processing*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1991.

Oppenheim, Alan V., and Schafer, Ronald W., *Digital Signal Processing*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1975.

Orfanidis, Sophocles J., *Introduction to Signal Processing*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1996.

Parks, T. W., and Burrus, C. S., *Digital Filter Design*, John Wiley & Sons, New York, 1987.

Proakis, John G., and Manolakis, Dimitris G., *Digital Signal Processing: Principles, Algorithms, and Applications*, 2nd ed., Macmillan Publishing Co., New York, 1992.

Programs for Digital Signal Processing, edited by Digital Signal Processing Committee of IEEE ASSP Society, IEEE Press, New York, 1979.

Roberts, Richard A., and Mullis, Clifford T., *Digital Signal Processing*, Addison-Wesley Publishing Co., Reading, MA, 1987.

Appendix B

Filter Design Software and C Code

WFILTER FILTER DESIGN SOFTWARE

WFilter is an analog and digital filter design package for Windows. To install WFilter on your computer, simply run the SETUP.EXE file in the \WFILTER directory on the accompanying CD.

WFilter determines the transfer function coefficients necessary for analog filters, and for digital FIR and IIR filters. The user is allowed to make choices of lowpass, highpass, bandpass, or bandstop filters for frequency selectivity as well as choices of approximation. For digital IIR and analog filters, the approximation choices are Butterworth, Chebyshev, inverse Chebyshev, and elliptic. For digital FIR filters, the rectangular, Barlett, Blackman, Hamming, von Hann, and Kaiser windows are available, as well as the Parks-McClellan optimization technique. The order of FIR filters based on design specifications cannot be predicted as accurately as for IIR and analog filters, therefore the user is given the option of changing the filter length during the design process.

After the filter has been designed, the user can view the pole-zero plot, as well as the magnitude and phase frequency responses. The filter design parameters or the frequency response parameters can also be edited for ease of use. Filter parameters can be saved and printed, and all plots can be printed or included in other documents by copying to the clipboard.

In addition, for analog filters, the Spice circuit file can be generated to aid in the analysis of active filters. After WFilter generated the file, it can be saved or printed. Digital filters may be used to filter wave files. After specifying an input wave file, a filtered output file can be generated, and both input and output files can be played (sound card must be present). In the case of an FIR filter, the user is given the option of filtering the sound file using convolution or the FFT.

C COMPUTER CODE

All of the C code discussed in the first eight chapters of this text (and much more) is included in the \C_Code directory on the CD that accompanies this text. The three short FFT functions discussed in Chapter 9 are listed in the text. The C code files originally accompanied the text *Analog and Digital Filter Design Using C* by Les Thede. There are three DOS executables that have been compiled by Microsoft Corp's Visual C++ compiler. The majority of WFilter is based on these files but with a GUI interface added for convenience.

FILTER.EXE - designs analog and digital filters.
ANALOG.EXE - implements analog active filters.
DIGITAL.EXE - implements digital IIR/FIR filters.

Three subdirectories have also been created on this disc to hold the source (.C), header (.H), and information (.TXT) files. In addition, sample sound files have been included to be used with the DIGITAL program.

Appendixes C through I discuss various C functions that are important parts of the filter design software.

FEEDBACK

I would appreciate any feedback you care to share about the text and software. Errors, problems, suggestions for improvement, and general comments can be forwarded to me via e-mail or the more traditional means. Thank you.

Les Thede - ECCS Department
Ohio Northern University
525 S. Main Street
Ada, Ohio 45810

Email: l-thede@onu.edu

Appendix C

Filter Design Using C

Although learning analog and digital filter design techniques is the first objective of this text, many readers may be interested in the use of C in the design and implementation of analog and digital filters. Therefore, supplementary material dealing with these programming issues is included in this and succeeding appendixes. All of the C code discussed in the first eight chapters of this text (and much more) is also included on the CD in the directory called \C_Code. The three short FFT functions in Chapter 9 are provided in the text.

The C programming language (and its successor C++) is a predominant force in engineering and computer science. In particular, C is the primary language used in digital filter design (with the possible exception of hardware-specific assembly language). C provides the combination of higher-level constructs while producing fast, compact executables. Other languages are available, and could be used in filter design, but C provides the best combination of efficiency and effectiveness. Several C language references are listed in Appendix A.

The primary data elements describing our filter will be used by a number of the design functions and therefore should be made available in a neat package. An array would be nice, but arrays require that all elements are of the same data type. Therefore the only reasonable choice is a structure.

We will store our filter data elements in a structure called `Filt_Params`. We can assume that we will need all of the filter specifications discussed in the first chapter. These will include the passband and stopband edge frequencies (`wpass1`, `wpass2`, `wstop1`, and `wstop2`) and gains (`apass1`, `apass2`, `astop1`, and `astop2`). We will also need indicators of the filter selectivity, the approximation method, and the implementation type (`select`, `approx`, and `implem`). The sampling frequency (`fsamp`) is an additional element that will be necessary for digital filters, but will not be used by analog filters. All of these variables represent the input specification for the filter design and are shown in the `Filt_Params` structure below.

```
typedef struct
{ double  apass1,apass2,  /*  passband gain's */
          astop1,astop2,  /*  stopband gain's */
          wpass1,wpass2,  /*  passband edge freq's */
          wstop1,wstop2,  /*  stopband edge freq's */
          fsamp,        /*  samp freq for dig filt */
          gain,         /*  gain multiplier */
          *acoefs,*bcoefs;  /*  ptr's to coefs */
  int     order;    /*  order or length of filter */
  /* selectivity, approximation and implementation */
  char    select,approx,implem;
} Filt_Params;
```

In addition to the input specifications, there will also be a need for another set of filter data once the filter has been designed. The filter's order (`order`) indicates the size of the filter's transfer function. The filter's gain constant (`gain`) and pointers to two sets of filter coefficients (`acoefs` and `bcoefs`) will completely describe the filter's transfer function and will be stored in the structure as type `double`.

It is impossible to determine the number of coefficients necessary to describe a filter before the filter is designed. We will design some filters with fewer than 10 coefficients, but other designs may require more than 200 coefficients. For that reason we will use pointers to the arrays instead of simply defining `acoefs` and `bcoefs` as large arrays. Using large arrays would make the structure unnecessarily large, and there would still be no guarantee that these fixed arrays would be large enough to store every filter that might be designed. It will be far more efficient to determine the size of the array necessary for the individual design, then allocate memory dynamically for the coefficient array, and finally store the pointer (address) to the array in the structure. That way only a pointer variable needs to be stored in the structure, not an entire array.

As it is, our `Filt_Params` structure is fairly large, and since we will be using this structure with most of the functions that we develop, it would be much more efficient to transfer a pointer (address) to the structure instead of the structure itself. Actually, there is another reason for doing this. In C all arguments of functions are “called by value,” which means that a copy of the argument is transferred to the function. This practice does not allow the transferred variable to be changed by the function. (Technically, we can change the variable all we want within the function, but when we leave the function the original value of the variable within the calling function remains unchanged.) Since our filter design functions will need to make changes to the data values stored in the structure, sending a copy of the structure to the function without allowing data values to change would be unacceptable. However, if we send a pointer to the structure (or actually a copy of the address), we will be able to access the actual memory locations of the data stored in the structure. In this way, our filter design functions will be able to enter values into the structure as necessary.

Appendix D

C Code for Normalized Approximation Functions

In the main file of the FILTER program on the CD, we called `Calc_Filter_Coefs` (as shown in Listing D.1) in order to calculate the necessary filter coefficients for the user-specified design. That function in turn called one of three other functions named `Calc_Analog_Coefs`, `Calc_DigFIR_Coefs`, or `Calc_DigIIR_Coefs`.

```
/*======================================================
  Calc_Filter_Coefs() - determines implementation and
  calls appropriate calculation function
  Prototype:  int Calc_Filter_Coefs(Filt_Params *FP);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
======================================================*/
int Calc_Filter_Coefs(Filt_Params *FP)
{ int Error;              /*  error value */
  /*  Call correct calc function for analog,
      digital FIR or IIR filters. */
  switch(FP->implem)
  { case 'A':
      Error = Calc_Analog_Coefs(FP);
      if(Error) { return 10*Error+1;}
      break;
    case 'F':
      Error = Calc_DigFIR_Coefs(FP);
      if(Error) { return 10*Error+2;}
      break;
    case 'I':
      Error = Calc_DigIIR_Coefs(FP);
      if(Error) { return 10*Error+3;}
      break;
    default:
      return ERR_FILTER;
  }
  return ERR_NONE;
}
```

Listing D.1 `Calc_Filter_Coefs` function.

The implementation type for the filter is stored in the `FP->implem` variable and is used to select an analog, digital FIR, or digital IIR filter implementation. In each case the user's specifications are transferred to the next appropriate function via `FP`, the pointer to the `Filt_Params` structure discussed in Appendix C. In this function, as well as others we will discuss, a simple error reporting system is used. Most of the functions used will return an integer value that will be zero if no error occurred and nonzero if an error did occur. At each new level of the program, the error value is multiplied by 10 and a single digit value is added. The result of this practice will be a multidigit error value where each digit represents a different level in the program. This can be used to trace the error to a particular function.

The `Calc_Analog_Coefs` function is given in Listing D.2. It is designed primarily to organize the calculation process, not to perform the calculations explicitly. The first function called is `Calc_Filter_Order`, which determines the filter order from the user specifications in `FP`. Next the normalized filter coefficients are determined using `Calc_Normal_Coefs`, which stores the coefficients in the `Filt_Params` structure. Finally, the coefficients are unnormalized to lowpass, highpass, bandpass, or bandstop coefficients at the user-specified frequencies via the `Unnormalize_Coefs` function.

```
/*======================================================
  Calc_Analog_Coefs() - calcs normal analog coefs
  Prototype:  int Calc_Analog_Coefs(Filt_Params *FP);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
======================================================*/
int Calc_Analog_Coefs(Filt_Params *FP)
{ int Error;                /*  error value */

  /*  Determine filter order, then normal coefs,
      then unnormalize them. */
  Error = Calc_Filter_Order(FP);
  if(Error) { return 10*Error+1;}
  Error = Calc_Normal_Coefs(FP);
  if(Error) { return 10*Error+2;}
  Error = Unnormalize_Coefs(FP);
  if(Error) { return 10*Error+3;}
  return ERR_NONE;
}
```

Listing D.2 `Calc_Analog_Coefs` function.

In the case of the `Calc_Normal_Coefs` function, as shown in Listing D.3, we must allow for individual functions to carry out the actual coefficient calculation. However, before we actually make the coefficient calculations, we allocate memory for the storage of the coefficients. By taking care of this necessary task in `Calc_Normal_Coefs`, we eliminate the need to handle it in each of the four individual functions. It is important to remember that by simply reserving a place in the `Filt_Params` structure for the pointers `acoefs` and `bcoefs`, we have *not* reserved any memory for the coefficients themselves. The pointers were originally set to zero or `NULL` by the `calloc` command we used to allocate memory for the `Filt_Params` structure. This effectively says that there is no memory available for coefficient storage, and if we try to access a coefficient while in this state, we would get an error. In fact, we hope we get an error to let us know that there is a problem in our algorithm. It would be far worse to have the pointers hold a nonzero value that is pointing to some random address in memory. In that case, we would get no error, but the values we would be accessing would be random nonsense.

```
/*====================================================
  Calc_Normal_Coefs() - allocates memory for coefs and
              calls proper function to calc coefs
  Prototype:  int Calc_Normal_Coefs(Filt_Params *FP);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
====================================================*/
int Calc_Normal_Coefs(Filt_Params *FP)
{ int Number_Coefs,   /*  Number of coefs in array */
      Error;          /*  error value */

  /*  Allocate memory for coefs.  There are 3 coefs
      for each quadratic. First-order factors are
      considered as quadratics. */
  Number_Coefs = 3 * ((FP->order + 1) / 2);
  FP->acoefs = (double *)
                malloc(Number_Coefs * sizeof(double));
  if(!FP->acoefs) { return ERR_ALLOC;}
  FP->bcoefs = (double *)
                malloc(Number_Coefs * sizeof(double));
  if(!FP->bcoefs) { return ERR_ALLOC;}
  /*  Calculate coefs based on approximation. */
  switch (FP->approx)
  { case 'B': Error = Calc_Butter_Coefs(FP);
              if(Error) { return 10*Error+1;}
              break;
    case 'C': Error = Calc_Cheby_Coefs(FP);
              if(Error) { return 10*Error+2;}
              break;
    case 'E': Error = Calc_Ellipt_Coefs(FP);
              if(Error) { return 10*Error+3;}
              break;
    case 'I': Error = Calc_ICheby_Coefs(FP);
              if(Error) { return 10*Error+4;}
              break;
    default:  return ERR_FILTER;
  }
  return ERR_NONE;
}
```

Listing D.3 `Calc_Normal_Coefs` function.

Up to this point in the program, we did not have the necessary information to determine the number of coefficients in each of the `acoefs` and `bcoefs` arrays. But since we now have the order of the filter, we can determine the number of coefficients exactly in the following manner. Each quadratic factor of the approximation function will require three coefficients (the s^2-term coefficient, the s-term coefficient, and the constant term coefficient). In addition, we will treat any odd-order approximation first-order factor as a quadratic. We can make use of the integer math in C to simplify the calculation for the number of coefficients based on the order of the filter.

Once the correct number of coefficients has been determined, `malloc` is used to allocate memory for that number of `double`s, and if an error occurs, we leave the function. The coefficient arrays are organized in the following manner. The `acoefs` array stores the coefficients for the numerator factors while the `bcoefs` array will store the coefficients for the denominator factors. (`acoefs` and `bcoefs` will take on different meanings in the digital filter design so for now we can remember A for above the line and B for below the line.) If the approximation function has an odd-order, the first-order coefficients are stored as a quadratic in the first three coefficients of each array with the s^2-term set to zero. The next three coefficients are for the first true quadratic, and then all other quadratics follow. If the approximation function has an even-order, then the first three coefficients are for the first quadratic, the next three coefficients for the second quadratic, and so on. Within the three coefficients, the first coefficient always refers to the s^2-term, the second refers to the s-term coefficients, and the last coefficient is the constant term in the quadratic factor.

The `Calc_Butter_Coefs` function shown in Listing D.4 is an example of the approximation calculation function. The function first checks for valid pointers and order values, and then proceeds to make the calculations outlined in Section D.2. If the order is odd, the first order coefficients are calculated followed by all coefficients for quadratic factors. The other individual functions for calculating the normalized coefficients for the four approximation methods can be found on the software disc in the \C_CODE\FILTER\F_DESIGN.C module. They all determine the coefficients in a manner consistent with our development earlier in this chapter.

There are a number of advanced math functions required by our approximation functions. These functions are *not* contained in F_DESIGN.C as are the rest of the functions discussed in this appendix, but are instead a part of the \C_CODE\FILTER\ADV_MATH.C module. These include the *asinh* and *acosh* functions used in the Chebyshev functions as well as the elliptic integral and Jacobian elliptic functions necessary to define the elliptic approximation. The values of these functions are typically calculated by the arithmetic-geometric mean method of iteration. A detailed discussion of this method and the elliptic functions in general is beyond the scope of this text, but references have been provided in the analog and digital filter design sections of Appendix A.

```
/*=====================================================
  Calc_Butter_Coefs() - calcs normal Butterworth coefs
  Prototype:  int Calc_Butter_Coefs(Filt_Params *FP);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
=====================================================*/
int Calc_Butter_Coefs(Filt_Params *FP)
{ int     m,a,b;         /*  Loop counter and indices*/
  double  R,epsilon,     /*  Intermediate values */
          theta,         /*  Angle location of poles */
          sigma,omega;   /*  Real/imag pos of poles */

  /*  Check for NULL ptrs and zero order. */
  if( !FP->acoefs)
  { return ERR_NULL;}
  if( !FP->bcoefs)
  { return ERR_NULL;}
  if(FP->order <= 0)
  { return ERR_VALUE;}
  /*  Make calculations of necessary constants. */
  epsilon = sqrt( pow(10.0,-0.1*FP->apass1) - 1.0 );
  R = pow(epsilon,-1.0/FP->order);
  /*  Initialize gain to 1.0. Start indices at 0 */
  FP->gain = 1.0;
  a = 0; b = 0;
  /*  Handle odd order if necessary. */
  if(FP->order % 2)
  { FP->acoefs[a++] = 0.0;
    FP->acoefs[a++] = 0.0;
    FP->acoefs[a++] = R;
    FP->bcoefs[b++] = 0.0;
    FP->bcoefs[b++] = 1.0;
    FP->bcoefs[b++] = R;
  }
  /*  Handle all quadratic terms. */
  for(m = 0;m < FP->order/2;m++)
  { /*  Calc angle first, then real and imag pos. */
    theta = PI*(2*m + FP->order +1) / (2 * FP->order);
    sigma = R * cos(theta);
    omega = R * sin(theta);
    /*  Set the quadratic coefs. */
    FP->acoefs[a++] = 0.0;
    FP->acoefs[a++] = 0.0;
    FP->acoefs[a++] = sigma*sigma+omega*omega;
    FP->bcoefs[b++] = 1.0;
    FP->bcoefs[b++] = -2 * sigma;
    FP->bcoefs[b++] = sigma*sigma+omega*omega;
  }
  return ERR_NONE;
}
```

Listing D.4 `Calc_Butter_Coefs` function.

The `Ellip_Integral` function is shown in Listing D.5 and uses the arithmetic-geometric mean method of determining the complete elliptic integral, as defined in (2.62). It takes as an argument the modulus k and returns the value of the complete elliptic integral. `MAX_TERMS` and `ERR_SMALL` have been defined as 100 and 1E-15, respectively, in the ADV_MATH.H include file. Of course these values could be changed as necessary.

```
/*======================================================
  Ellip_Integral() - calcs complete elliptic integral
              using arithmetic-geometric mean method
  Prototype:  void Ellip_Integral(double k);
  Return:     complete elliptic integral value
  Arguments:  k - the modulus of the integral
======================================================*/
double Ellip_Integral(double k)
{ int     i;      /*  Loop counter. */
  double  A[MAX_TERMS],B[MAX_TERMS],
          C[MAX_TERMS]; /* Array storage values. */

  /*  Square the modulus as required by this method.*/
  k = k * k;
  /*  Initialize the starting values. */
  A[0] = 1;
  B[0] = sqrt(1-k);
  C[0] = sqrt(k);
  /*  Iterate until error is small enough. */
  for(i = 1; i < MAX_TERMS ;i++)
  { A[i] = (A[i-1] + B[i-1])/2;
    B[i] = sqrt(A[i-1]*B[i-1]);
    C[i] = (A[i-1] - B[i-1])/2;
    if(C[i] < ERR_SMALL)
    { break;}
  }
  return PI / (2 * A[i]);
}
```

Listing D.5 `Ellip_Integral` function.

Appendix E

C Code for Unnormalized Approximation Functions

The `Unnormalize_Coefs`, which is given in Listing E.1, first determines the variables used for unnormalization. The unnormalization frequency `freq` for lowpass and highpass, as well as the center frequency `Wo` and the bandwidth `BW` for bandpass and bandstop cases are determined differently for the inverse Chebyshev case as compared to the other approximation methods. After these calculations, the appropriate unnormalization function is chosen based on the selectivity of the filter. Each of the specific functions called uses the `Filt_Params` structure pointer `FP` as well as the appropriate unnormalization variables. Any errors that occur in the functions are handled in the manner described in Appendix D to allow easy identification of the location of the problem.

Listing E.2 contains the `Unnorm_LP_Coefs`, which handles the lowpass unnormalization. The function first determines whether there is a first-order factor by determining if the order of the approximation is odd. Remembering our technique of always placing first-order factors in the coefficient arrays first, we can safely refer to the constant term coefficients using the index of 2. (The coefficients are arranged with the s^2-term coefficient first, then the s-term coefficient, and finally the constant term coefficient.) Each additional quadratic factor then follows in the same order. The coefficients within the loop are adjusted as we determined in Section 3.1. Proper indexing is accomplished by using `cf` to index individual coefficients based on `qd`, the quadratic indicator. Using this technique, any coefficient using `cf` as an index is referring to an s^2-term coefficient, while if the coefficient has an index of `cf+1`, it is an s-term coefficient, and `cf+2` will be the index for the constant term of a quadratic expression. Note that we are using the efficient C style where `a *= b;` is equivalent to `a = a * b;`.

The `Unnorm_HP_Coefs` function is very similar to the previous function and therefore will not be discussed here. That function can be found in the \C_CODE\FILTER\F_DESIGN.C module.

```
/*====================================================
  Unnormalize_Coefs() - converts normal lowpass coefs
              to unnormalized LP/HP/BP/BS.
  Prototype:  int Unnormalize_Coefs(Filt_Params *FP);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
====================================================*/
int Unnormalize_Coefs(Filt_Params *FP)
{ int     Error;    /*  error value */
  double  freq, /*  unnormalizing freq for LP & HP  */
          BW,   /*  unnormal. bandwidth for BP & BS */
          Wo;   /*  unnormal. ctr freq for BP & BS  */

  /*  Calc freq, Wo and BW based on approx method */
  switch(FP->approx)
  { case 'B':
    case 'C':
    case 'E':
      freq = FP->wpass1;
      Wo = sqrt(FP->wpass1 * FP->wpass2);
      BW = FP->wpass2 - FP->wpass1;
      break;
    case 'I':
      freq = FP->wstop1;
      Wo = sqrt(FP->wstop1 * FP->wstop2);
      BW = FP->wstop2 - FP->wstop1;
      break;
    default:
      return ERR_FILTER;
  }
  /*  Call unnormal. function based on selectivity  */
  switch(FP->select)
  { case 'L':
      Error = Unnorm_LP_Coefs(FP,freq);
      if(Error) { return 10*Error+1;}
      break;
    case 'H':
      Error = Unnorm_HP_Coefs(FP,freq);
      if(Error) { return 10*Error+2;}
      break;
    case 'P':
      Error = Unnorm_BP_Coefs(FP,BW,Wo);
      if(Error) { return 10*Error+3;}
      break;
    case 'S':
      Error = Unnorm_BS_Coefs(FP,BW,Wo);
      if(Error) { return 10*Error+4;}
      break;
    default:
      return ERR_FILTER;
  }
  return ERR_NONE;
}
```

Listing E.1 `Unnormalize_Coefs` function.

```
/*======================================================
  Unnorm_LP_Coefs() - converts normal lowpass coefs to
              unnormal LP coefs at a specific freq.
  Prototype:  int Unnorm_LP_Coefs(Filt_Params *FP,
                                          double freq);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
              freq - unnormalization frequency
======================================================*/
int Unnorm_LP_Coefs(Filt_Params *FP,double freq)
{ int qd,cf,          /*  quad and coef number */
      qd_start;       /*  starting quad for loop */


  /*  Handle first-order, if odd; set qd_start */
  if(FP->order % 2)
  { FP->acoefs[2] *= freq;
    FP->bcoefs[2] *= freq;
    qd_start = 1;
  }
  else
  { qd_start = 0;}


  /*  Handle quadratic factors, qd indexes through
      quadratic factors, cf converts to coef number */
  for(qd = qd_start; qd < (FP->order + 1)/2; qd++)
  { cf = qd * 3;
    FP->acoefs[cf+1] *= freq;
    FP->acoefs[cf+2] *= (freq * freq);
    FP->bcoefs[cf+1] *= freq;
    FP->bcoefs[cf+2] *= (freq * freq);
  }
  return ERR_NONE;
}
```

Listing E.2 `Unnorm_LP_Coefs` function.

Listing E.3 shown below gives the `Unnorm_BP_Coefs` function. This function and the `Unnorm_BS_Coefs` function are more complicated than the lowpass and highpass functions. One of the ways that these functions are more complicated is that the coefficient arrays must be resized to hold the larger number of coefficients for a bandpass function. (Remember that for bandpass and bandstop filters, the final order will be twice the original lowpass normalized order.) Thus, early in this function, variables from the original lowpass function are stored as well as the new order of the bandpass function. Then new pointers to the larger arrays of bandpass coefficients are assigned while leaving the original pointers unchanged. Throughout this function `new_num`, `new_den`, `org_num` and `org_den` are used to identify the new and original numerator and denominator coefficients, respectively. As in previous functions, we handle the first-order factors before the second-order factors. The values assigned to the new coefficients are the same as we determined in Section 3.3.

```
/*======================================================
  Unnorm_BP_Coefs() - converts normal lowpass coefs to
              unnormal BP coefs at a specific freq.
  Prototype:  int Unnorm_BP_Coefs(Filt_Params *FP,
                                double BW,double Wo);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
              BW - bandwidth for unnormalization
              Wo - center freq for unnormalization
======================================================*/
int Unnorm_BP_Coefs(Filt_Params *FP,
                                  double BW,double Wo)
{ int     qd,ocf,ncf,qd_start,/* loop cntrs, indexes*/
          numb_coefs,         /* num coefs in array */
          org_quads,          /* orig num of quads  */
          org_order;          /* original order     */
  double  *org_num,*org_den,  /* orig num, den ptrs */
          *new_num,*new_den;  /* new num, den  ptrs */
  complex A,B,C,D,E;          /* temp cmplx vars    */

  /*  Store orig number of quads and order,
      new order will be twice original.  */
  org_order = FP->order;
  org_quads = (org_order + 1)/2;
  FP->order = org_order * 2;
  /*  For clarity, assign ptrs to temp variables  */
  org_num = FP->acoefs;
  org_den = FP->bcoefs;
  /*  Three coefs for each new quad=3*(new_order+1)/2,
      but new_order will be even, so its simplified */
  numb_coefs = 3 * org_order;
  /*  Allocate memory for new arrays with more coefs*/
  new_num=(double *)malloc(numb_coefs*sizeof(double));
  if(!new_num)  { return ERR_ALLOC;}
  new_den=(double *)malloc(numb_coefs*sizeof(double));
  if(!new_den)  { return ERR_ALLOC;}
  /*  If org_order odd, convert first-order factor to
      quadratic, qd_start indic start pt for loop */
  if(org_order % 2)
  { new_num[0] = org_num[1];
    new_num[1] = BW * org_num[2];
    new_num[2] = org_num[1] * Wo * Wo;
    new_den[0] = org_den[1];
    new_den[1] = BW * org_den[2];
    new_den[2] = org_den[1] * Wo * Wo;
    qd_start = 1;
  }
  else
  { qd_start = 0;}
  /*  Each orig quad term will be converted to two new
      quads via complex quadratic factoring. */
  for(qd = qd_start;qd < org_quads;qd++)
  { /*  ocf indexes org coefs, 3 coefs per org quad
        ncf indexes new coefs, 6 coefs per org quad
        ncf also adjusts for first-order factor */
    ocf = qd * 3;
    ncf = qd * 6 - qd_start * 3;
    /*  For numers which DON'T have s^2 or s terms. */
    if(org_num[ocf] == 0.0)
    { new_num[ncf] = 0.0;
      new_num[ncf+1] = sqrt(org_num[ocf+2]) * BW;
```

```
        new_num[ncf+2] = 0.0;
        new_num[ncf+3] = 0.0;
        new_num[ncf+4] = sqrt(org_num[ocf+2]) * BW;
        new_num[ncf+5] = 0.0;
      }
      /*  For numers which DO have s^2 and s terms. */
      else
      { /*  Convert coefs to complex, then factor */
        A = cmplx(org_num[ocf],0);
        B = cmplx(org_num[ocf+1],0);
        C = cmplx(org_num[ocf+2],0);
        cQuadratic(A,B,C,&D,&E);
        /*  Make required substitutions, factor again */
        A = cmplx(1,0);
        B = cmul(cneg(D),cmplx(BW,0));
        C = cmplx(Wo*Wo,0);
        cQuadratic(A,B,C,&D,&E);
        /*  Determine final values for new coefs. */
        new_num[ncf] = 1.0;
        new_num[ncf+1] = -2.0 * creal(D);
        new_num[ncf+2] = creal(cmul(D,cconj(D)));
        new_num[ncf+3] = 1.0;
        new_num[ncf+4] = -2.0 * creal(E);
        new_num[ncf+5] = creal(cmul(E,cconj(E)));
      }
      /*  Denoms will always have nonzero s^2 term. */
      /*  Convert coefs to complex, then factor */
      A = cmplx(org_den[ocf],0);
      B = cmplx(org_den[ocf+1],0);
      C = cmplx(org_den[ocf+2],0);
      cQuadratic(A,B,C,&D,&E);
      /*  Make required substitutions, factor again */
      A = cmplx(1,0);
      B = cmul(cneg(D),cmplx(BW,0));
      C = cmplx(Wo*Wo,0);
      cQuadratic(A,B,C,&D,&E);
      /*  Make required substitutions, factor again */
      new_den[ncf] = 1.0;
      new_den[ncf+1] = -2.0 * creal(D);
      new_den[ncf+2] = creal(cmul(D,cconj(D)));
      new_den[ncf+3] = 1.0;
      new_den[ncf+4] = -2.0 * creal(E);
      new_den[ncf+5] = creal(cmul(E,cconj(E)));
    }
    /*  Free the memory allocated to original coefs. */
    free(FP->acoefs);
    free(FP->bcoefs);
    /*  Assign the new ptrs to old array ptrs. */
    FP->acoefs = new_num;
    FP->bcoefs = new_den;
    return ERR_NONE;
  }
```

Listing E.3 `Unnorm_BP_Coefs` function.

Before we begin the discussion of the unnormalization of second-order factors within the `for` loop, we need to make a slight excursion into the use of complex numbers in C. The C language does not support complex numbers as a standard data type as it does `int`s, `float`s, and `double`s. Therefore, if we are to use them in the solution of the unnormalization of bandpass and bandstop coefficients, we will have to define our own complex number definition. A complex number can easily be defined with a structure using a real and imaginary member as shown below.

```
typedef struct
{ double  re,   /*  Real part of complex number */
          im;   /*  Imag part of complex number */
}   complex;
```

Using this `complex struct` will allow us to define any variables we like as type `complex`. For example, in the variable declaration section of the function we are studying now, the variables `A`, `B`, `C`, `D`, and `E` are defined as `complex` as shown below.

```
complex A,B,C,D,E;          /* temp cmplx vars    */
```

Within the \C_CODE\FILTER\COMPLEX.C module there are a number of complex functions defined to implement standard mathematical functions for complex numbers. A list of the functions is shown below. We do not have space to study them here, but the full module is contained on the software disc included with this text.

```
cadd() – adds complex numbers and returns result.
cang() – returns angle (radians) of a complex number.
cconj() – returns complex conjugate of a complex number.
cdiv() – divides complex numbers and returns result.
cimag() – returns imaginary part of a complex number.
cmag() – returns the magnitude of a complex number.
cmplx() – returns complex number made from two doubles.
cmul() – multiplies complex numbers and returns result.
cneg() – returns the negative of a complex number.
cprt() – prints the value of a complex number.
cQuadratic() – factors quadratic eqn. with complex coefficients.
creal() – returns real part of a complex number.
csqr() – returns the square root of a complex number.
csub() – subtracts complex numbers and returns result.
```

As an example of one of these complex functions, `cadd` is shown in Listing E.4. As indicated in the listing, `cadd` takes two complex numbers as arguments and adds their respective real and imaginary parts. The new complex number `x` is then returned to the calling function.

Another complex function is shown in Listing E.5 below. `cQuadratic` takes five arguments, all of which are complex or point to complex numbers. The first three arguments, `a`, `b`, and `c`, are the coefficients of a quadratic equation, while `d` and `e` are the addresses where the factors of the quadratic equation are to be stored. Each line of the function makes a partial calculation of the quadratic formula, but using complex functions. The final results are then stored where the variables `d` and `e` point. Remember this advanced version of the quadratic equation solver is necessary because some of the coefficients of the quadratic equation are complex, unlike the typical situation where they all would be real.

```
/*======================================================
  cadd() - adds complex numbers (a+b), returns result
======================================================*/
complex cadd(complex a,complex b)
{ complex x;
  x.re = a.re + b.re;
  x.im = a.im + b.im;
  return x;
}
```

Listing E.4 `cadd` function.

```
/*======================================================
  cQuadratic() - solves quadratic equation with cmplx
    coefficients. Equation form is a*x^2 + b*x + c,
    solutions will be placed in cmplx numbers d and e,
    whose addresses are sent to cQuadratic.
======================================================*/
void cQuadratic(complex a,complex b,complex c,
                                complex *d,complex *e)
{ complex a2,ac4,sq;  /*  intermediate values */

  a2 = cmul(a,cmplx(2,0));          /*  2*a         */
  ac4 = cmul(cmul(a,c),cmplx(4,0)); /*  4*a*c       */
  sq = csqr(csub(cmul(b,b),ac4)); /* sqrt(b*b-4*a*c)*/
  *d = cdiv(cadd(cneg(b),sq),a2);   /*  first root  */
  *e = cdiv(csub(cneg(b),sq),a2);   /*  second root */
}
```

Listing E.5 `cQuadratic` function.

Now we can return to the discussion of the `Unnorm_BP_Coefs`. `qd_start` is again used to control the starting point of the `for` loop, and `org_quads` controls the ending point. Once we enter the loop to unnormalize the original second-order factors we first define indexing variables to control the location within the original coefficient array and the new coefficient array. The variable `ocf` controls the position within the original coefficient array by indexing three positions for each original quadratic factor. The variable `ncf` controls the position within the new coefficient array by indexing six positions for each of the original quadratics. The six positions are necessary because for each original quadratic there will be two new quadratics produced. In addition `ncf` must adjust for the unnormalized first-order factor if there is one. Therefore, `ncf` starts at 0 if the original order was even, but starts at 3 if the order was odd. Thereafter, it increments by a value of 6.

The numerator unnormalization can be of two different types. In the Butterworth and Chebyshev cases, there will be no s^2-term or s-term, while the inverse Chebyshev and elliptic approximations will have an s^2-term present for the complex zeros. Therefore, an `if` statement is used to determine the appropriate method to use for a particular case. In the first case, only *s*-term coefficients take on the nonzero values we determined in Section 3.3. In the second case, all coefficients take on nonzero values, which are dependent on a number of complex calculations.

In this second, more complicated case, we must first convert the original coefficients into complex numbers using the `cmplx` function. Then the roots of the quadratic are determined by the `cQuadratic` function and stored in the variables `D` and `E`. (We will not be dealing with the `E` root of this first quadratic because we know that it is the complex conjugate of the `D` root.) We are then ready to define another quadratic as we found in Section 3.3. The appropriate values from this quadratic are loaded into `A`, `B`, and `C`, and then `cQuadratic` is called again. The roots of this second quadratic are then stored in `D` and `E` again. Each of these roots will define one quadratic, just as one pole location in Chapter 2 was enough to determine a quadratic. For example, if $D = \alpha + j\beta$ and $E = \delta + j\lambda$, then the D root will produce a quadratic of the forms $s^2 - 2\,\alpha\, s + (\alpha^2 + \beta^2)$ and the E root will produce $s^2 - 2\,\delta\, s + (\delta^2 + \lambda^2)$. These representations are mirrored in the C code. Note that the sum of squares is calculated by multiplying the root by its complex conjugate. The denominator quadratics are calculated in just the same manner as this numerator case.

Once we leave the loop, all of the new coefficients have been calculated and put in place. All we have left to do is to reassign the array pointers in the `Filt_Params` structure. We are now finished with the original coefficients from the lowpass normalized approximation, so we can free the memory allocated to them by using the `free` command. Next, we take the pointers for the new, larger arrays and put them into `FP`. Now the pointers in `FP` point to areas in memory that store the larger arrays of bandpass coefficients, and the areas in memory that stored the original coefficients have been freed for future use.

At this point, we have covered all of the necessary description for `Unnorm_BS_Coefs` as well. There are really no significant differences in the bandstop unnormalization function. It can be found in the \C_CODE\FILTER\F_DESIGN.C module.

Appendix F

C Code for Active Filter Implementation

In order to aid in the implementation and evaluation of analog active filters, we will now develop the code necessary for the calculation of component values and the generation of PSpice text files. These text files will serve as input to PSpice that will analyze the active filters.

We will use two structures to pass information to the various functions in this project. The first structure is the familiar `Filt_Params` structure, discussed previously. The second is a new structure used to hold the component values for the active filters to be designed. This `RC_Comps` structure is shown below and includes the voltage divider resistors R_x and R_y as well as the number of stages in the filter. The other variables contained in the structure are pointers to arrays because these variables will be present in each stage of the filter and we will not know at the time of the program execution how many stages will be present. The components included are the primary R and C values, the gain control resistors R_A and R_B, and the bandstop parameters R_o and C_o.

```
typedef struct
{ double  Rx,Ry,    /*  Voltage divider values */
          *R,*C,    /*  Primary R-C values */
          *Ra,*Rb,  /*  Feedback resistors */
          *Ro,*Co;  /*  Twin-tee addl components */
  int     stages;   /*  Number of stages */
} RC_Comps;
```

Once we have the common values of capacitor C_O and resistor R_A to use in each of the filter's stages, as well as the frequency information, we can calculate the other circuit values using the `Calc_Components` function. (This function, as well as others associated with analog filter implementation, can be found in \C_CODE\ANALOG\ANALOG.C.) `Calc_Components` will in turn call an appropriate function based on the selectivity of the desired filter. For example, if a lowpass filter is being implemented, then the `Calc_LP_Comps` function will be called, while if a bandpass filter is being implemented, the `Calc_BP_Comps` function will be called. However, no matter which function is to be called, memory must be allocated for the arrays of component values in the structure.

This memory allocation can be done prior to calling the individual functions and thereby eliminate the requirement of memory allocation in each of the individual functions.

The component calculation functions are very similar to one another although the equations for component values are somewhat different. For that reason, only the `Calc_LP_Comps` function shown in Listing F.1 will be discussed here. The work within the function begins by initializing the variable `K_Total`, which will store the product of all stage *K*s. Then, if a first-order stage is required, the *R* and *C* values for it are calculated and `start` is set to 1. Then, the component values for the stages implementing the quadratic factors are calculated within the `for` loop, which has a starting index of `start`. The coefficients from the `Filt_Params` structure are retrieved, and the common values of *C* and R_A are placed in the `RC_Comps` structure. Then a determination has to be made within a switch statement as to whether a filter with complex conjugate zeros will be required. If the zeros are not required, the calculations are made and the process continues. However, if complex conjugate zeros are required, it must be decided whether a resistor, capacitor, or no additional value is necessary.

The loop will continue its iterations until all stage components have been determined. Once the loop finishes, the gain adjustment factor can be determined and the voltage divider components can be calculated. If the gain adjustment is exactly 1, no values are calculated since a division by zero would result.

```
/*======================================================
  Calc_LP_Comps() - calculates the component values
  for lowpass analog active filter.
  Prototype:  int Calc_LP_Comps(Filt_Params *FP,
         RC_Comps *RC,double C,double Ra);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
              RC - ptr to struct holding RC components
              C -  capacitor value for all stages
              Ra - feedback resistor for all stages
======================================================*/
int Calc_LP_Comps(Filt_Params *FP,RC_Comps *RC,
                                   double C,double Ra)
{ int     i,start;    /* Loop counter and start pt*/
  double  K,K_Total,  /* K value and total */
          Gain_Adj,   /* Gain adjustment factor */
          a2,a2r,     /* Numerator constants */
          b1,b2,b2r;  /* Denominator constants */

  /*  Initialize K total */
  K_Total = 1;
  /*  If order is odd, determine R & C values for
      first-order stage and set start to 1 */
  start = 0;
  if(FP->order % 2)
  { RC->C[0] = C;
    RC->R[0] = 1 / (C * FP->bcoefs[2]);
    RC->Ra[0] = Ra;
    start = 1;
  }
  /*  Determine values for second-order stages */
```

```
    for(i = start; i < RC->stages ;i++)
    { /*  Determine coefficients and roots */
      a2 = FP->acoefs[i*3 + 2];
      a2r = sqrt(a2);
      b1 = FP->bcoefs[i*3 + 1];
      b2 = FP->bcoefs[i*3 + 2];
      b2r = sqrt(b2);
      /*  Set standard values in structure */
      RC->C[i] = C;
      RC->Ra[i] = Ra;
      /*  Calculate values based on approx type
          B,C use Sallen-Key, E,I use Twin-Tee */
      switch(FP->approx)
      { case 'B':
        case 'C':
          RC->R[i] = 1 / (C * b2r);
          K = 3 - (b1/b2r);
          break;
        case 'E':
        case 'I':
          RC->R[i] = 1 / (C * a2r);
          /*  Find K, Ro, Co dependent on a2,b2 */
          if(a2 > b2)
          {       K = 2 + ((a2-b2-b1*a2r)/(2*b2));
            RC->Co[i] = (((a2/b2) - 1) * C) / 2;
          }
          else if(a2 < b2)
          {       K = 2 + ((b2-a2-b1*a2r)/(2*a2));
            RC->Ro[i] = 2 * RC->R[i] / ((b2/a2) - 1);
          }
          else
          { K = 2 - (b1/(2*a2r));}
          break;
        default:
          return ERR_FILTER;
      }
      /*  If Co = 0, K_Tot only increases by K */
      K_Total *= ( (K * C) / (C + 2*RC->Co[i]) );
      RC->Rb[i] = Ra * (K - 1);
    }
    /*  Make final adjustment of gain and
        calculate voltage divider values */
    Gain_Adj = K_Total / FP->gain;
    if(Gain_Adj == 1.000)
    { return ERR_NONE;}
    RC->Rx = Gain_Adj * R_OUT;
    RC->Ry = Gain_Adj * R_OUT / (Gain_Adj - 1);
    return ERR_NONE;
  }
```

Listing F.1 `Calc_LP_Comps` function.

After an analog active filter has been designed and the component values have been calculated, the next logical step is to test the circuit. Testing usually includes both computer analysis, where a circuit simulation is performed, and laboratory analysis, where the circuit is built from components and tested with electronic equipment. We can help in the computer evaluation of the filter circuit by preparing the analysis data file necessary for PSpice tool.

The `Write_Circ_File` function contained in ANALOG.C is used to coordinate the generation of the circuit analysis text file. After all of the filter sections have been written, the final voltage divider section is appended, and an appropriate model for the op-amp circuit is specified. Finally, the analysis modes are specified using the starting and ending frequencies provided by the user. As an example of one of the functions that generates circuit analysis data files, Listing F.2 shows `Write_LP_Section`.

```
/*======================================================
  Write_LP_Section() - writes a lowpass filter
  section to the circuit data file.
  Prototype:  void Write_LP_Section(int stage,
                          RC_Comps *RC,FILE *CF);
  Return:     none
  Arguments:  stage - section number of filter
              RC - ptr to struct holding RC components
              CF - ptr to output file
======================================================*/
void Write_LP_Section(int stage,RC_Comps *RC,
                                        FILE *CF)
{ int     s,t;        /* Stage related variables */
  double  R,C,Ra,Rb;  /* Component values */

  /*  Simplify some variables */
  t = stage + 1;                       s = 10 * t;
  R = RC->R[stage];   C = RC->C[stage];
  Ra = RC->Ra[stage]; Rb = RC->Rb[stage];
  /*  Rb == 0 if first-order stage, otherwise
      generate circuit text for second-order */
  if(Rb == 0)
  { fprintf(CF,"\n*    Stage Number %d",stage+1);
    fprintf(CF,"\nR%d\t%d\t%d\t%8.3E",s+1,s+1,s+2,R);
    fprintf(CF,"\nC%d\t%d\t%d\t%8.3E",s+1,s+2,0,C);
    fprintf(CF,"\nRb%d\t%d\t%d\t1",t,s+3,s+11);
    fprintf(CF,
        "\nX%d\t%d\t%d\t%d\tOPAMP",t,s+2,s+3,s+11);
  }
  /*  Generate circuit text for second-order stage
      If Ro and Co != 0, then use BS stage to
      generate elliptic or inv Chebyshev approx,
      otherwise use standard BP configuration */
  else
  {if( (RC->Co[stage] != 0) || (RC->Ro[stage] != 0) )
   { Write_BS_Section(stage,RC,CF);}
   else
   {fprintf(CF,"\n*    Stage Number %d",stage+1);
    fprintf(CF,"\nR%d\t%d\t%d\t%8.3E",s+1,s+1,s+2,R);
    fprintf(CF,"\nR%d\t%d\t%d\t%8.3E",s+2,s+2,s+3,R);
    fprintf(CF,"\nC%d\t%d\t%d\t%8.3E",s+1,s+3,0,C);
    fprintf(CF,"\nC%d\t%d\t%d\t%8.3E",s+2,s+2,s+11,C);
    fprintf(CF,"\nRa%d\t%d\t%d\t%8.3E",t,s+4,0,Ra);
    fprintf(CF,"\nRb%d\t%d\t%d\t%8.3E",t,s+4,s+11,Rb);
    fprintf(CF,"\nX%d\t%d\t%d\t%d\tOPAMP",t,s+3,s+4,s+11);
   }
  }
}
```

Listing F.2 `Write_LP_Section` function.

This function begins by simplifying the form of the components for each stage and then writes a section based on the order of the filter stage and the implementation of the filter stage. If the stage is implementing a first-order factor as indicated by R_B having a value of zero, the necessary component values are written to the file. If the stage is implementing a second-order factor, the stage configuration could be either a Sallen-Key or a twin-tee notch form. If either the capacitor C_o or the resistor R_o is nonzero, the twin-tee notch form is indicated, and the `Write_BS_Section` function is called since it implements the twin-tee notch filter form. Otherwise, the components for a standard Sallen-Key stage are written to the text file.

Appendix G

C Code for IIR Filter Design

The bilinear transform approach to IIR filter design is popular because it has wide application. Our procedure will be the same as outlined in Chapter 6: prewarp the critical digital frequencies to their analog counterparts, design a standard analog filter using the new specifications, perform the bilinear transform on the analog transfer function to produce the digital transfer function, and finally, set the frequencies back to their original values for future use. The design of the analog filter has already been covered in earlier chapters, so we won't need to develop that code. Consequently, our work involves generating only a few new functions in order to add the IIR filter design capability to our project.

In the `Calc_DigIIR_Coefs` function shown in Listing G.1, we first use `Warp_Freqs` to prewarp the frequencies; then, we call `Calc_Filter_Order`, `Calc_Normal_Coefs`, and `Unnormalize_Coefs` in order to design the analog filter. After the analog filter has been designed, the analog coefficients can be converted to digital coefficients by the `Bilinear_Transform` function. And, finally, the `UnWarp_Freqs` function converts the critical frequencies back to their original values. Throughout this process, all critical parameters are transferred to and from the function in the `Filt_Params` structure using the pointer `FP`.

During the filter specification phase of our program, the user entered the desired frequency specifications that the digital filter must satisfy. In the `Warp_Freqs` function, these frequencies must be converted to analog frequencies using the techniques discussed in Chapter 6. We can combine (6.7) and (6.18) in order to determine the relationship that must exist between the analog and digital radian frequencies. The `UnWarp_Freqs` function simply resets the critical frequencies using the relationship of (G.2).

$$\omega_a = \frac{2}{T} \cdot \tan\left(\frac{2 \cdot \pi \cdot f_d}{2 \cdot f_s}\right) = 2 \cdot f_s \cdot \tan\left(\frac{\omega_d}{2 \cdot f_s}\right) \qquad \text{(G.1)}$$

$$\omega_d = 2 \cdot f_s \cdot \tan^{-1}\left(\frac{\omega_a}{2 \cdot f_s}\right) \tag{G.2}$$

```
/*======================================================
  Calc_DigIIR_Coefs() - calcs digital IIR coefs
  Prototype:  int Calc_DigIIR_Coefs(Filt_Params *FP);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
======================================================*/
int Calc_DigIIR_Coefs(Filt_Params *FP)
{ int Error;        /*  error value */

  /*  Pre-warp frequencies before making calcs */
  Error = Warp_Freqs(FP);
  if(Error) { return 10*Error+1;}
  /*  Calc order and coefs, then unnormalized coefs */
  Error = Calc_Filter_Order(FP);
  if(Error) { return 10*Error+2;}
  Error = Calc_Normal_Coefs(FP);
  if(Error) { return 10*Error+3;}
  Error = Unnormalize_Coefs(FP);
  if(Error) { return 10*Error+4;}
  /*  Transform from s-domain to z-domain */
  Error = Bilinear_Transform(FP);
  if(Error) { return 10*Error+5;}
  /*  Put the critical freqs back to orig value */
  Error = UnWarp_Freqs(FP);
  if(Error) { return 10*Error+6;}
  return ERR_NONE;
}
```

Listing G.1 `Calc_DigIIR_Coefs` function.

The objective of the `Bilinear_Transform` function shown in Listing G.2 is to implement the transformation from an analog transfer function to a digital transfer function. A check that the `acoefs` and `bcoefs` arrays actually exist is made first in the function. Then, constants that will be used often are calculated and stored in `f2` and `f4`. And, finally, the number of quadratics is calculated as well as a starting point for the quadratic loop. If the order of the filter is odd, then a first-order term is handled, and `start` is set to 1. Notice that `start` controls which quadratic factor will start the process. If a first-order factor (which is stored as a quadratic) has already been processed, the `for` loop will start with the second quadratic (if one is present). The actual transformation calculations are handled in exactly the same manner as derived in (6.20) to (6.25). Also notice that the total gain of the filter is adjusted in the first-order case as well as in the quadratic loop. By the end of the function, all gain adjustments have been included and the analog coefficients have been replaced by digital IIR coefficients.

```
/*=====================================================
  Bilinear_Transform() - use bilinear transform to
  convert transfer function from s-domain to z-domain
  Prototype:  int Bilinear_Transform(Filt_Params *FP,
                                      double fsamp);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
=====================================================*/
int Bilinear_Transform(Filt_Params *FP)
{ int     i,j,start, /*  loop counters and index */
          num_quads; /*  number of quad factors */
  double  f2,f4,     /*  2 * fsamp, and 4 * fsamp^2 */
          N0,N1,N2,  /*  numerator temp variables */
          D0,D1,D2;  /*  denominator temp variables */
  if( (!FP->acoefs) || (!FP->bcoefs) )
  { return ERR_NULL;}
  /*  determine some constants  */
  f2 = 2 * FP->fsamp;
  f4 = f2 * f2;
  num_quads = (FP->order + 1)/2;
  /*  handle first-order factor if present  */
  start = 0;
  if(FP->order % 2)
  { N0 = FP->acoefs[2] + FP->acoefs[1] * f2;
    N1 = FP->acoefs[2] - FP->acoefs[1] * f2;
    D0 = FP->bcoefs[2] + FP->bcoefs[1] * f2;
    D1 = FP->bcoefs[2] - FP->bcoefs[1] * f2;
    FP->acoefs[0] = 1.0;
    FP->acoefs[1] = N1 / N0;
    FP->acoefs[2] = 0.0;
    FP->bcoefs[0] = 1.0;
    FP->bcoefs[1] = D1 / D0;
    FP->bcoefs[2] = 0.0;
    FP->gain *= (N0 / D0);
    start = 1;
  }
  /*  Handle quadratic factors. */
  for(i = start; i < num_quads ;i++)
  { j = 3 * i;
    N0 = FP->acoefs[j]*f4 + FP->acoefs[j+1]*f2 + FP->acoefs[j+2];
    N1 = 2 * (FP->acoefs[j+2] - FP->acoefs[j]*f4);
    N2 = FP->acoefs[j]*f4 - FP->acoefs[j+1]*f2 + FP->acoefs[j+2];
    D0 = FP->bcoefs[j]*f4 + FP->bcoefs[j+1]*f2 + FP->bcoefs[j+2];
    D1 = 2 * (FP->bcoefs[j+2] - FP->bcoefs[j]*f4);
    D2 = FP->bcoefs[j]*f4 - FP->bcoefs[j+1]*f2 + FP->bcoefs[j+2];
    FP->acoefs[j] = 1.0;
    FP->acoefs[j+1] = N1 / N0;
    FP->acoefs[j+2] = N2 / N0;
    FP->bcoefs[j] = 1.0;
    FP->bcoefs[j+1] = D1 / D0;
    FP->bcoefs[j+2] = D2 / D0;
    FP->gain *= (N0 / D0);
  }
  return ERR_NONE;
}
```

Listing G.2 `Bilinear_Transform` function.

Appendix H

C Code for FIR Filter Design

The design of digital FIR filters is accomplished by `Calc_DigFIR_Coefs` shown in Listing H.1. (All of the functions necessary to implement that section of the project can be found in the \C_CODE\FILTER\F_DESIGN.C module on the software disc.) In this function, we see that all of the necessary steps in the design process are accomplished by the functions called from `Calc_DigFIR_Coefs`. In addition, the user is given an opportunity to adjust the length of the filter after it has been estimated. This option is necessary since the FIR filter length cannot be calculated exactly. Once the length has been accepted or changed, memory can be allocated for the filter coefficients.

The length of the FIR filter is estimated by the `Estm_Filter_Len` function. In this function the various parameters required to estimate the length of either a window FIR design or a Parks-McClellan design are calculated. All window designs use the Kaiser estimate, which provides a starting point for the filter designer. In most cases, the Kaiser will be the preferred window design with the other methods used for comparison purposes. After the filter length has been estimated, the calculated value is converted to the next higher odd integer and stored in `FP->order.` We recognize that the variable is actually the filter's length, but this saves us from defining another variable in the `Filt_Params` structure.

If the design method uses the window technique, the ideal filter coefficients are calculated using `Calc_Ideal_FIR_Coefs.` This function implements the appropriate equation for ideal coefficient calculation and stores them in the FP–>bcoefs array. Next, the proper window coefficients are calculated using one of several `Calc_xxxx_Win_Coefs` functions and stored in the `FP–>acoefs` array. Finally, the ideal and window coefficients are multiplied by the `Multi_Win_Ideal_Coefs` function with the final coefficients stored in the `FP–>acoefs`.

```
/*=====================================================
  Calc_DigFIR_Coefs() - calcs the digital FIR coefs
  Prototype:  int Calc_DigFIR_Coefs(Filt_Params *FP);
  Return:     error value
  Arguments:  FP - ptr to struct holding filter params
=====================================================*/
int Calc_DigFIR_Coefs(Filt_Params *FP)
{ char ans;
  int Error;        /*  error value */
  double beta;      /*  parameter for Kaiser window */

  /*  Estimate the length (order) of filter.
      Get beta for Kaiser window, if needed. */
  beta = Estm_Filter_Len(FP);
  /*  See if user wants to adjust estimated length */
  printf("\n Filter length is estimated as %d.", FP->order);
  ans = Get_YN("\n Do you wish to change it? (Y/N):");
  if(ans == 'Y')
  { FP->order = Get_Int("\n Please enter new length: ",0,500);}
  /*  Allocate memory for coefficients. */
  FP->acoefs =
        (double *) malloc(FP->order * sizeof(double));
  if(!FP->acoefs) { return ERR_ALLOC;}
  FP->bcoefs = (double *) malloc(FP->order * sizeof(double));
  if(!FP->bcoefs) { return ERR_ALLOC;}
  /*  Set overall gain to 1.0 */
  FP->gain = 1.0;
  /*  Calculate the ideal FIR coefficients
      but not for Parks-McClellan. */
  if(FP->approx != '6')
  { Error = Calc_Ideal_FIR_Coefs(FP);
    if(Error) { return 10*Error+1;}
  }
  /*  Determine the approximation method to use.  */
  switch(FP->approx)
  { case '0': Error = Calc_Rect_Win_Coefs(FP);
              if(Error) { return 10*Error+2;}  break;
    case '1': Error = Calc_Bart_Win_Coefs(FP);
              if(Error) { return 10*Error+3;}  break;
    case '2': Error = Calc_Blck_Win_Coefs(FP);
              if(Error) { return 10*Error+4;}  break;
    case '3': Error = Calc_Hamm_Win_Coefs(FP);
              if(Error) { return 10*Error+5;}  break;
    case '4': Error = Calc_Hann_Win_Coefs(FP);
              if(Error) { return 10*Error+6;}  break;
    case '5': Error = Calc_Kais_Win_Coefs(FP,beta);
              if(Error) { return 10*Error+7;}  break;
    case '6': Error = Calc_ParkMccl_Coefs(FP);
              if(Error) { return 10*Error+8;}  break;
    default:  return ERR_FILTER;
  }
  /*  Multiply window and ideal coefs only for
      but not for Parks-McClellan coefficients */
  if(FP->approx != '6')
  { Error = Mult_Win_Ideal_Coefs(FP);
    if(Error) { return 10*Error+9;}
  }
  return ERR_NONE;
}
```

Listing H.1 `Calc_DigFIR_Coefs` function.

Appendix I

Filtering Sound Files

There are a number of sound file formats in use today, but one of the most popular is the WAVE file format (.WAV). This file format has a number of different ways that the file information can be stored, but we will concentrate on just the basic techniques. We discuss only the formats for monaural and stereo signals with either 8 bits or 16 bits per sample. Compression schemes are popular today to save space in transferring or saving music files, but we will concentrate only on uncompressed files. We will see that handling four different options will provide us with enough challenge for now.

Each sound file begins with a header of information that describes the important characteristics of the file such as sampling frequency, number of samples, number of channels (mono or stereo) and number of bits per sample. For our work, the header information for each file is shown in Table I.1.

After the header information, the raw data for the sound file is provided in one of the four formats. If the data file is monaural, the data is just a sequence of bytes or integers depending on the number of bits per sample. The number of data *values* can be determined from the information in the header, which specifies the number of data *bytes* in the file. If the file is using 16 bits (2 bytes) per sample, then the number of data values is one-half of the number of data bytes. In the case of a stereo sound file, the byte or integer samples of each channel are alternated starting with the left channel and then the right channel. Therefore, if we need to know how many samples to process for the left channel of a 16 bits per sample stereo sound file, we would need to divide the number of data bytes by four to arrive at the proper value.

If the data is in the form of 16 bits per sample, then we can treat the data as a simple signed integer that has a range of values from +32,767 to –32,768. If the data is in the form of 8 bits per sample, then it is stored as an unsigned character with values from 0 to 255 with 128 considered as the midpoint. This is *not* the same as a signed character data type, and therefore special consideration must be given to the conversion of 8-bit to 16-bit representation. Equation (I.1) indicates the proper procedure to convert from 8-bit unsigned data to 16-bit signed data, while (I.2) shows the opposite conversion. (The functions `Convert2Char` and

`Convert2Int` are included in the \C_CODE\DIGITAL directory of the accompanying software disc as are all other functions associated with digital filter implementation.)

Table I.1

File Format for .WAV Files

Bytes	*Description*
0 – 3	“RIFF” — identification string
4 – 7	Reserved
8 – 15	“WAVEfmt∅“ — ID string (∅ = space)
16 – 19	Reserved
20 – 21	Type of format — short integer
22 – 23	Number of channels — short integer
24 – 27	Samples per second — long integer
28 – 31	Average bytes per second — long integer
32 – 33	Block alignment — short integer
34 – 35	Bits per sample — short integer
36 – 39	“data” — identification string
40 – 43	Number of data bytes — long integer

$$\text{data}_{16} = (\text{data}_8 - 128) \cdot 256 \tag{I.1}$$

$$\text{data}_8 = (\text{data}_{16} / 256) + 128 \tag{I.2}$$

We are now ready to discuss the functions necessary to filter a sound file. We will need to determine the parameters of the filter as well as the waveform to be filtered. Then we will need to read in the waveform data, filter it, and write out the processed data. Since there are four different formats that could be used, and since we don’t want to generate four different filtering algorithms to handle each one, we standardize each file type into a monaural 16-bit data waveform. Then, the filtering algorithms can be optimized to operate on that type of file. Since we will be operating in a nonreal-time mode, this conversion should not be a problem.

In the case of reading the input data and writing the output data, we must handle the conversion of 8 bits per sample and stereo files. If the file uses 8 bits per sample, the input data waveform is first converted to 16 bits per sample. If the input file is stereo, we then separate it into two monaural files and process them as two independent files. The filtering process will be relatively easy since we have already developed the functions to accomplish this in Chapter 8. A special structure is used to store information about the waveform to be processed. The

Wvfrm_Params structure contains all of the important information about the waveform.

```
typedef struct
{ char  *header;          /* ptr to header info */
  int   file_type,        /* type of data file */
        numb_chan,        /* number of channels */
        bytes_per_samp;   /* bytes per sample */
  long  samp_per_sec,     /* samples per second */
        numb_samples;     /* number of samples */
} Wvfrm_Params;
```

Listing I.1 Wvfrm_Params structure.

The Digital_Filter function performs most of the work in the program and is quite long. Therefore, only an abbreviated version is shown in Listing I.2. Array creation, error checking, and conversions from 8 to 16 bit as well as stereo to mono versions have been removed. (The complete function can be viewed on the software disc.) The actual digital filtering is handled by using one of the functions discussed in earlier sections. Based on the filter implementation type and the status of the REAL_TIME constant (defined in DIGITAL.H), we will use Dig_IIR_Filter, Dig_FIR_Filt_RT, or Dig_FIR_Filt_NRT to produce the filtering.

```
/*=====================================================
  Digital_Filter() - determines the type of waveform
  (mono/stereo - 8bit/16bit), type of filter (FIR/IIR)
  and sets up all memory for filtering process
  Prototype:  int Digital_Filter(FILE *InFile,
      FILE *OutFile,Filt_Params *FP,Wvfrm_Params *WP);
  Return:     error value.
  Arguments:  InFile - input data file
              OutFile - output data file
              FP - ptr to Filt_Params struct
              WP - ptr to Wvfrm_Params struct
=====================================================*/
int Digital_Filter(FILE *InFile,FILE *OutFile,
                     Filt_Params *FP,Wvfrm_Params *WP)
{ /* Declaration of variables (not shown - NS) */

  /*  Set all pointers to null */
  a = c = C = M1 = M2 = 0;
  X1 = X2 = Y1 = Y2 = 0; Z = 0;
  /*  Set common values */
  bytes = WP->bytes_per_samp;
  chan = WP->numb_chan;
  error = ERR_NONE;

  /* Allocate memory for data (NS) */

  /*  ===== ===== Handle the IIR case ===== =====  */
  if(FP->implem == 'I')
  { /*  Set numb_quads and start for later use */
    numb_quads = (FP->order + 1) / 2;
    start = 0;
```

```
        /*  Alloc memory for input, coefs and mem (NS) */
        /*  Set up second set of arrays if stereo (NS) */
        /*  Load the coef array with gain, b's and a's */
        c = C;
        *c++ = FP->gain;
        for(i = 0; i < numb_quads ;i++)
        { j = i * 3;
          *c++ = FP->bcoefs[j+1];
          *c++ = FP->bcoefs[j+2];
          *c++ = FP->acoefs[j+1];
          *c++ = FP->acoefs[j+2];
        }
      }
      /*  ===== ===== Handle the FIR case ===== =====  */
      else if(FP->implem == 'F')
      { /*  Set numb_coefs and start for later use */
        numb_coefs = FP->order;
        if(REAL_TIME)
        { start = 0;}
        else
        { start = numb_coefs - 1;}
        /*  Alloc memory for input, coefs and mem (NS) */
        /*  Set up second set of array if stereo (NS) */
        }
        /*  Load the coef array (in reverse order)
            with a's and gain */
        c = C;
        a = FP->acoefs + numb_coefs - 1;
        for(i = 0; i < numb_coefs ;i++)
        { *c++ = *a--;}
        *c++ = FP->gain;
      }
      else
      { error = ERR_VALUE; goto TIDY_UP;}
      k = 0;
      /*  Start outer loop of filtering process */
      Total = WP->numb_samples * WP->numb_chan;
      Done = 0;
      while(!Done)
      { /* Read in data */
        numb_read = fread(&X1[start],sizeof(int),
                                CHUNK_SIZE * chan,InFile);
        /*  Select IIR or FIR filter for ch 1 or mono */
        if(FP->implem == 'I')
        { Dig_IIR_Filter(X1,Y1,M1,C,numb_quads,numb_read/chan);}
        if(FP->implem == 'F')
        { if(REAL_TIME)
          { Dig_FIR_Filt_RT(X1,Y1,M1,C,numb_coefs,numb_read/chan);}
          else
          { Dig_FIR_Filt_NRT(X1,Y1,C,numb_coefs,numb_read/chan);}
        }
        /*  Write out data */
        numb_writ = fwrite(Y1,sizeof(int),numb_read,OutFile);
      } /*  End of while loop */
      TIDY_UP:
      /*  Free memory, close files and return (NS) */
      return error;
    }
```

Listing I.2 Abbreviated `Digital_Filter` file.

About the Author

Les Thede is a professor of electrical and computer engineering at Ohio Northern University, Ada, Ohio. A former design engineer for Motorola, Inc., he holds a B.S and M.S. in electrical engineering from the University of Iowa and a Ph.D. in engineering science from the University of Toledo. He established the DSP lab at Ohio Northern University in 1989 and has written several articles on filter design and C programming. He has written a previous book on analog and digital filter design and currently is teaching courses in filter design, digital signal processing, and image processing. He is a member of IEEE and ASEE. His e-mail address is l-thede@onu.edu.

Index