Chapter

12

FIR Filter Design: Frequency Sampling Method

12.1 Introduction

In Chap. 11, the desired frequency response for an FIR filter was specified in the continuous-frequency domain, and the discrete-time impulse response coefficients were obtained via the Fourier series. We can modify this procedure so that the desired frequency response is specified in the discrete-frequency domain and then use the inverse discrete Fourier transform (DFT) to obtain the corresponding discrete-time impulse response.

Example 12.1 Consider the case of a 21-tap lowpass filter with a normalized cutoff frequency of $\lambda_U = 3\pi/7$. The sampled magnitude response for positive frequencies is shown in Fig. 12.1. The normalized cutoff frequency λ_U falls midway between $n = 4$ and $n = 5$, and the normalized folding frequency of $\lambda = \pi$ falls midway between $n = 10$ and $n = 11$. (Note that $45/10.5 = 3/7$.) We assume that $H_d(-n) = H_d(n)$ and use the inverse DFT to obtain the filter coefficients listed in Table 12.1. The actual continuous-frequency

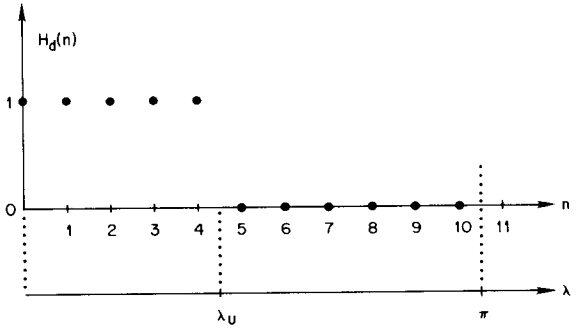

Figure 12.1 Desired discrete-frequency magnitude response for a lowpass filter with $\lambda_U = 3\pi/7$.

TABLE 12.1 Coefficients for the 21-tap Filter of Example 12.1

$h[0] = h[20] =$	0.037334
$h[1] = h[19] =$	−0.021192
$h[2] = h[18] =$	−0.049873
$h[3] = h[17] =$	0.000000
$h[4] = h[16] =$	0.059380
$h[5] = h[15] =$	0.030376
$h[6] = h[14] =$	−0.066090
$h[7] = h[13] =$	−0.085807
$h[8] = h[12] =$	0.070096
$h[9] = h[11] =$	0.311490
$h[10] =$	0.428571

response of an FIR filter having these coefficients is shown in Figs. 12.2 and 12.3. Figure 12.2 is plotted against a linear ordinate, and dots are placed at points corresponding to the discrete-frequencies specified in Fig. 12.1. Figure 12.3 is included to provide a convenient baseline for comparison of subsequent plots that will have to be plotted against decibel ordinates in order to show low stop-band levels.

The ripple performance in both the pass-band and stop-band responses can be improved by specifying one or more transition-band samples at values somewhere between the pass-band value of $H_d(m) = 1$ and the stop-band

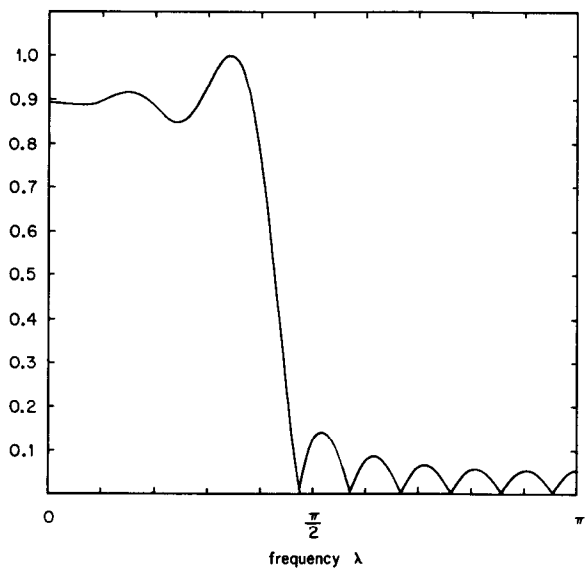

Figure 12.2 Magnitude response for filter of Example 12.1.

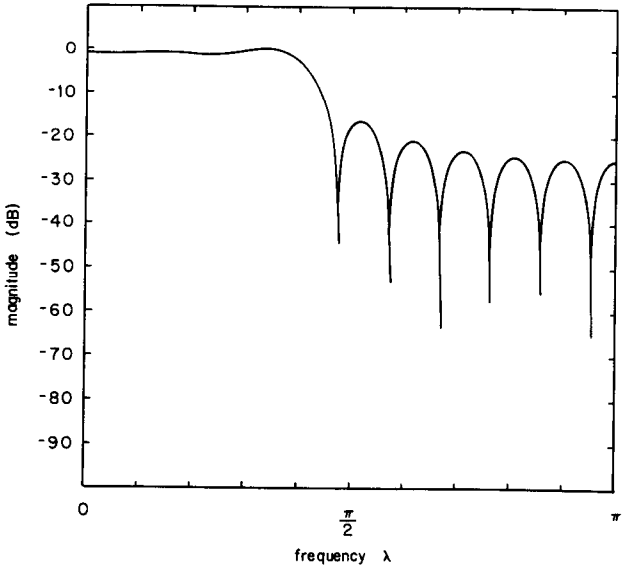

Figure 12.3 Filter response for Example 12.1 plotted on decibel scale.

value of $H_d(m) = 0$. Consider the case depicted in Fig. 12.4 where we have modified the response of Fig. 12.1 by introducing a one-sample transition band by setting $H_d(5) = 0.5$. The continuous-frequency response of this modified filter is shown in Fig. 12.5, and the coefficients are listed in Table 12.2.

The peak stop-band ripple has been reduced by 13.3 dB. An even greater reduction can be obtained if the transition-band value is optimized rather than just arbitrarily set halfway between the pass-band and the stop-band levels. It is also possible to have more than one sample in the transition band. The methods for optimizing transition-band values are iterative and involve repeatedly computing sets of impulse response coefficients and the corresponding frequency responses. Therefore, before moving on to specific optimization approaches, we will examine some of the mathematical details and

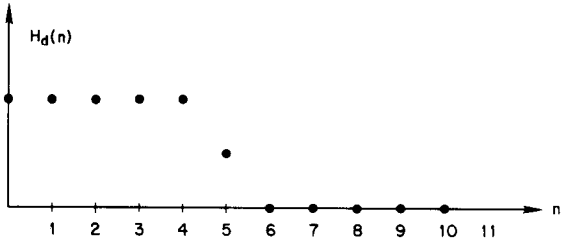

Figure 12.4 Discrete-frequency magnitude response with one transition-band sample midway between the ideal pass-band and stop-band levels.

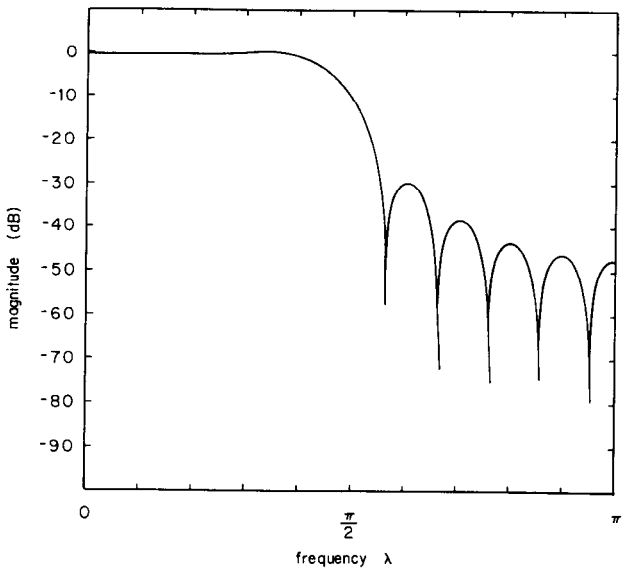

Figure 12.5 Continuous-frequency magnitude response corresponding to the discrete-frequency response of Fig. 12.3.

TABLE 12.2 Coefficients for the 21-tap Filter with a Single Transition-Band Sample Value of 0.5

$h[0] = h[20] =$	0.002427
$h[1] = h[19] =$	0.008498
$h[2] = h[18] =$	−0.010528
$h[3] = h[17] =$	−0.023810
$h[4] = h[16] =$	0.016477
$h[5] = h[15] =$	0.047773
$h[6] = h[14] =$	−0.020587
$h[7] = h[13] =$	−0.096403
$h[8] = h[12] =$	0.023009
$h[9] = h[11] =$	0.315048
$h[10] =$	0.476190

explore some ways for introducing some computational efficiency into the process.

12.2 Odd *N* versus Even *N*

Consider the desired response shown in Fig. 12.6 for the case of an odd-length filter with no transition band. If we assume that the cutoff lies midway

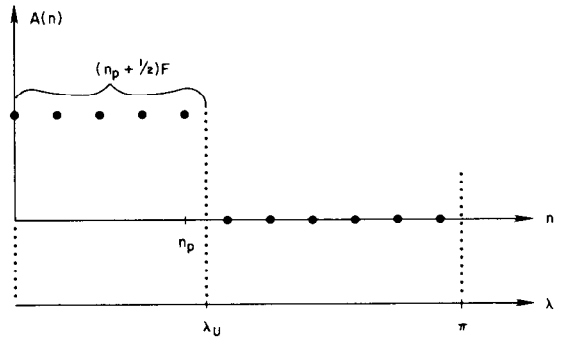

Figure 12.6 Desired frequency-sampled response for an odd-length filter with no transition-band samples.

between $n = n_p$ and $n = n_p + 1$ as shown, the cutoff frequency is $2\pi F(n_p + \frac{1}{2})$, where F is the interval between frequency domain samples. For the normalized case where $T = 1$, we find $F = 1/N$, so the normalized cutoff is given by

$$\lambda_U = \frac{\pi(2n_p + 1)}{N} \tag{12.1}$$

This equation allows us to compute the cutoff frequency when n_p and N are given. However, in most design situations we will need to start with known (desired) values of N and λ_U and then determine n_p. We can solve (12.1) for n_p, but for an arbitrary value λ_U, the resulting value of n_p might not be an integer. Therefore, we write

$$n_p = \left\lfloor \frac{N\lambda_{UD}}{2\pi} - \frac{1}{2} \right\rfloor \tag{12.2}$$

where λ_{UD} denotes desired λ_U and $\lfloor \cdot \rfloor$ denotes the "floor" function that truncates the fractional part from its argument. Equation (12.2) yields a value for n_p that guarantees that the cutoff will lie *somewhere* between n_p and $n_p + 1$, but not necessarily at the midpoint. The difference $\Delta\lambda = |\lambda_U - \lambda_{UD}|$ is an indication of how "good" the choices of n_p and N are—the smaller $\Delta\lambda$ is, the better the choices are.

It is a common practice to assume that the cutoff frequency lies midway between $n = n_p$ and $n = n_p + 1$ as in the preceding analysis. If the continuous-frequency amplitude response is a straight line between $A(n) = 1$ at $n = n_p$ and $A(n) = 0$ at $n = n_p + 1$, the value of the response midway between these points will be 0.5. However, since $A(n)$ is the *amplitude* response, the attenuation at the assumed cutoff is 6 dB. For an attenuation of 3 dB, the cutoff should be assigned to lie at a point which is 0.293 to the right of n_p and 0.707 to the left of $n_p + 1$.

If we assume that the cutoff lies at $n_p + 0.293$, the cutoff frequency is $2\pi F(n_p + 0.293)$ and the normalized cutoff is given by

$$\lambda_U = \frac{2\pi(n_p + 0.293)}{N} \tag{12.3}$$

The required number of samples in the (two-sided) pass band is $2n_p + 1$ where

$$n_p = \left\lfloor \frac{N\lambda_{UD}}{2\pi} - 0.293 \right\rfloor$$

For convenience we will denote the λ_U given by (12.1) as λ_6 and the λ_U given by (12.3) as λ_3.

Even *N*

Now let's consider the response shown in Fig. 12.7 for the case of an even-length filter with no transition band. If we assume that the cutoff lies midway between $n = n_p$ and $n = n_p + 1$, the cutoff frequency is $2\pi F n_p$ and the normalized cutoff is

$$\lambda_6 = \frac{2\pi n_p}{N}$$

Solving for n_p and using the floor function to ensure integer values, we obtain

$$n_p = \left\lfloor \frac{N\lambda_{6D}}{2\pi} \right\rfloor$$

If we assume that the cutoff lies at $n_p + 0.293$, the cutoff frequency is $2\pi F(n_p - 0.207)$ and the normalized cutoff is

$$\lambda_3 = \frac{2\pi(n_p - 0.207)}{N}$$

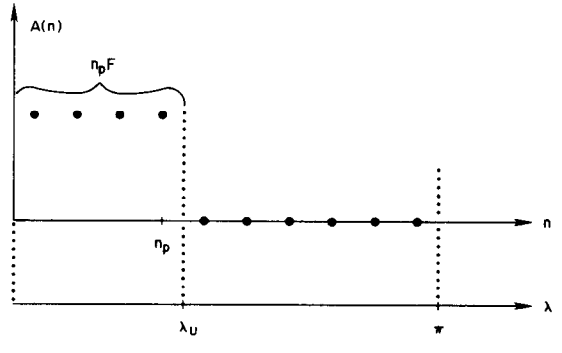

Figure 12.7 Desired frequency-sampled response for an even-length filter with no transition-band samples.

The required number of samples in the (two-sided) pass band $2n_p$ where

$$n_p = \left\lfloor \frac{N\lambda_{3D}}{2\pi} + 0.207 \right\rfloor$$

If processing constraints or other implementation considerations place an upper limit $N_{\max}$ on the total number of taps that can be used in a particular situation, it might be smart to choose between $N = N_{\max}$ and $N = (N_{\max} - 1)$ based upon which value of N yields λ_U that is closer to λ_{UD}.

Example 12.2 For $N_{\max} = 21$ and $\lambda_{6D} = 3\pi/7$, determine whether $N = 21$ or $N = 20$ would be the better choice based on values of $\Delta\lambda$.

solution For $N = 20$,

$$n_p = \left\lfloor \frac{20[(3\pi/7)]}{2\pi} \right\rfloor = \left\lfloor \frac{30}{7} \right\rfloor = 4$$

$$\lambda_6 = \frac{2\pi(4)}{20} = \frac{2\pi}{5}$$

$$\Delta\lambda = \left| \frac{3\pi}{7} - \frac{2\pi}{5} \right| = \frac{\pi}{35}$$

For $N = 21$,

$$n_p = \left\lfloor \frac{21[(3\pi/7)]}{2\pi} - \frac{1}{2} \right\rfloor = \lfloor 4 \rfloor = 4$$

$$\lambda_6 = \frac{9\pi}{21} = \frac{3\pi}{7}$$

$$\Delta\lambda = \left| \frac{3\pi}{7} - \frac{3\pi}{7} \right| = 0$$

For this contrived case, $N = 21$ is not only the better choice—it is the best choice, yielding $\Delta\lambda = 0$.

Example 12.3 For $N_{\max} = 21$ and $\lambda_{3D} = 2\pi/5$, determine whether $N = 21$ or $N = 20$ would be the better choice based on values of $\Delta\lambda$.

solution For $N = 20$,

$$n_p = \left\lfloor \frac{20[(2\pi/5)]}{2\pi} + 0.207 \right\rfloor = \lfloor 4.209 \rfloor = 4$$

$$\lambda_3 = \frac{2\pi(4 - 0.207)}{20} = 1.1916$$

$$\Delta\lambda = \left| \frac{2\pi}{5} - 1.1916 \right| = 0.065$$

For $N = 21$,

$$n_p = \left\lfloor \frac{21[(2\pi/5)]}{2\pi} - 0.293 \right\rfloor = \lfloor 3.907 \rfloor = 3$$

$$\lambda_3 = \frac{2\pi(3.293)}{21} = 0.9853$$

$$\Delta\lambda = \left| \frac{2\pi}{5} - 0.9853 \right| = 0.2714$$

Since $0.065 < 0.2714$, the better choice appears to be $N = 20$.

12.3 Design Formulas

The inverse DFT can be used as it was in Example 12.1 to obtain the impulse response coefficients $h(n)$ from a desired frequency response that has been specified at uniformly spaced discrete frequencies. However, for the special case of FIR filters with constant group delay, the inverse DFT can be modified to take advantage of symmetry conditions. Back in Sec. 8.2, the DTFT was adapted to the four specific types of constant-group-delay FIR filters to obtain the dedicated formulas for $H(\omega)$ and $A(\omega)$ that were summarized in Table 10.1. For the discrete-frequency case, the DFT can be similarly adapted to obtain the explicit formulas for $A(k)$ given in Table 12.3. (The entries in the table are for the normalized case where $T = 1$.) After some trigonometric manipulation, we can arrive at the corresponding inverse relations or *design formulas* listed in Table 12.4. These formulas are implemented by the C function **fsDesign( )** provided in Listing 12.1.

TABLE 12.3 Discrete-Frequency Amplitude Response of FIR Filters with Constant Group Delay

Type	
1 $h[n]$ symmetric N odd	$h[M] + \sum_{n=0}^{M-1} 2h[n] \cos\left[\frac{2\pi(M-n)k}{N}\right] = h[M] + \sum_{n=1}^{M} 2h[M-n] \cos\left(\frac{2\pi kn}{N}\right)$
2 $h[n]$ symmetric N even	$\sum_{h=0}^{(N/2)-1} 2h[n] \cos\left[\frac{2\pi(M-n)k}{N}\right] = \sum_{n=1}^{N/2} 2h\left[\frac{N}{2}-n\right] \cos\left\{\frac{2\pi k[n-(1/2)]}{N}\right\}$
3 $h[n]$ antisymmetric N odd	$\sum_{n=0}^{M-1} 2h[n] \sin\left[\frac{2\pi(M-n)k}{N}\right] = \sum_{n=1}^{M} 2h[M-n] \sin\left(\frac{2\pi kn}{N}\right)$
4 $h[n]$ antisymmetric N even	$\sum_{n=0}^{(N/2)-1} 2h[n] \sin\left[\frac{2\pi(M-n)k}{N}\right] = \sum_{n=1}^{N/2} 2h\left[\frac{N}{2}-n\right] \sin\left\{\frac{2\pi k[n-(1/2)]}{N}\right\}$

TABLE 12.4 Formulas for Frequency Sampling Design of FIR Filters with Constant Group Delay

Type	$h[n]$ $n = 0, 1, 2, \ldots, N-1$
1 $h[n]$ symmetric N odd	$\frac{1}{N}\left\{A(0) + \sum_{k=1}^{M} 2A(k)\cos\left[\frac{2\pi(n-M)k}{N}\right]\right\}$
2 $h[n]$ symmetric N even	$\frac{1}{N}\left\{A(0) + \sum_{k=1}^{(N/2)-1} 2A(k)\cos\left[\frac{2\pi(n-M)k}{N}\right]\right\}$
3 $h[n]$ antisymmetric N odd	$\frac{1}{N}\left\{\sum_{k=1}^{M} 2A(k)\sin\left[\frac{2\pi(M-n)k}{N}\right]\right\}$
4 $h[n]$ antisymmetric N even	$\frac{1}{N}\left\{A\left(\frac{N}{2}\right)\sin[\pi(M-n)] + \sum_{k=1}^{(N/2)-1} 2A(k)\sin\left[\frac{2\pi(M-n)k}{N}\right]\right\}$

12.4 Frequency Sampling Design with Transition-Band Samples

As mentioned in the introduction to this chapter, the inclusion of one or more samples in a transition band can greatly improve the performance of filters designed via the frequency sampling method. In Sec. 12.1, some improvement was obtained by simply placing one transition-band sample halfway between the pass band's unity amplitude and the stop band's zero value. However, even more improvement can be obtained if the value of this single transition-band sample is "optimized." Before proceeding, we need to first decide just what constitutes an "optimal" value for this sample—we could seek the sample that minimizes pass-band ripple, minimizes stop-band ripple, or minimizes some function that depends upon both stop-band and pass-band ripple. The most commonly used approach is to optimize the transition-band value so as to minimize the peak stop-band ripple.

For any given set of desired amplitude response samples, determination of the peak stop-band ripple entails the following steps:

1. From the specified set of desired amplitude response samples H_d, compute the corresponding set of impulse response coefficients h using the C function **fsDesign( )** presented in Sec. 12.3.
2. From the impulse response coefficients generated in step 1, compute a fine-grained discrete-frequency approximation to the continuous-frequency amplitude response using the C function **cgdFirResponse( )** presented in Sec. 10.3.
3. Search the amplitude response generated in step 2 to find the peak value in the stop band. This search can be accomplished using the C function **findSbPeak( )** given in Listing 12.2.

In general, we will need five parameters to specify the location of the stop band(s) so that **findSbPeak( )** "knows" where to search. The first parameter specifies the band configuration—lowpass, highpass, bandpass, or bandstop. The other parameters are indices of the first and last samples in the filter's pass bands and stop bands. Lowpass and highpass filters need only two parameters n_1 and n_2, but bandpass and bandstop filters need four: n_1, n_2, n_3, and n_4. The specific meaning of these parameters for each of the basic filter configurations is shown in Fig. 12.8. For easier argument passing, **findSbPeak( )** has been designed to expect the filter configuration specified in a single input array **bandConfig[ ]** as follows:

bandConfig[0] = 1 for lowpass, 2 for highpass,
3 for bandpass, 4 for bandstop

bandConfig[1] = n_1

bandConfig[2] = n_2

bandConfig[3] = n_3

bandConfig[4] = n_4

bandConfig[5] = number of taps in filter

To see how this information is used, consider the lowpass case where n_2 is the index of the first stop-band sample in the desired response $H_d[n]$. The goal is to find the peak stop-band value in the filter's *continuous-frequency* magnitude response. The computer must compute samples of a discrete-frequency approximation to this continuous-frequency response. This approximation should not be confused with the desired response $H_d[n]$, which is also a discrete-frequency magnitude response. The latter contains only N samples, where N is the number of taps in the filter. The approximation to the continuous-frequency response must contain a much larger number of points. The number of samples in the (one-sided) approximation to the continuous response is supplied to **findSbPeak( )** as the integer argument **numPts**. For the examples in this chapter, values for **numPts** ranging from 120 to 480 have been used. In searching for the peak of a lowpass response, **findSbPeak( )** directs its attention to samples n_s and beyond in the discrete-frequency approximation to the continuous-frequency amplitude response where

$$n_s = \frac{2Ln_2}{N}$$

and L = number of samples in the (one-sided) approximation to the continuous response (that is, **numPts**)
N = number of taps in the filter
n_2 = index of first sample in the desired (positive-frequency) stop band

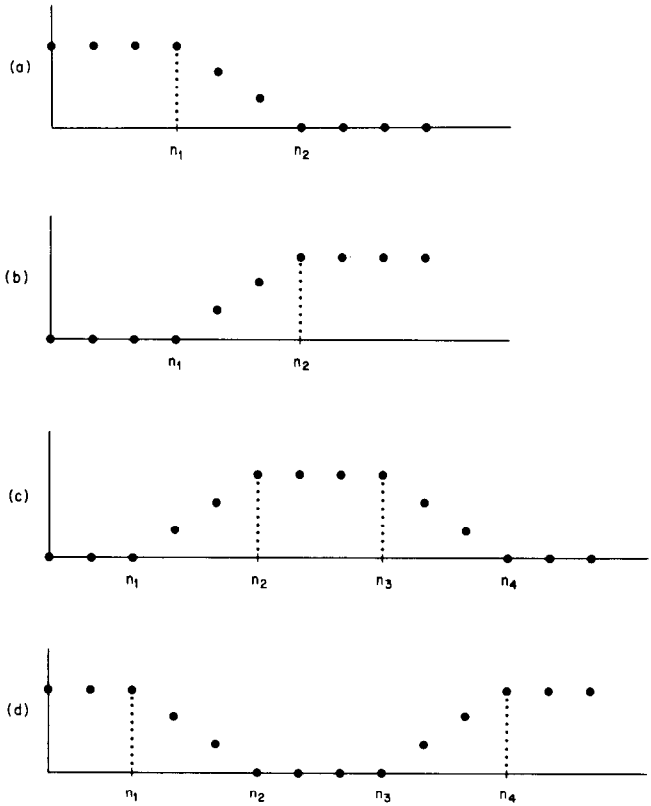

Figure 12.8 Parameters for specifying band configurations: (*a*) lowpass, (*b*) highpass, (*c*) bandpass, and (*d*) bandstop.

For highpass, bandpass, and bandstop filters, the search is limited to the stop band in a similar fashion.

The approach for finding the peak, as outlined in steps 1 through 3 above, contains some "fat" that could be eliminated to gain speed at the expense of clarity and modularity. For example, computing the entire amplitude response is not necessary, since only the stop-band values are of interest to the optimization procedure. Also, for any given filter, consecutive peaks in the response will be separated by a number of samples that remains more or less constant—this fact could be exploited to compute and examine only those portions of the response falling within areas where stop-band ripple peaks can be expected.

Optimization

In subsequent discussions, T_A will be used to denote the value of the single transition-band sample. One simple approach for optimizing the value of T_A

is to just start with $T_A = 1$ and keep decreasing by some fixed increment, evaluating the peak stop-band ripple after each decrease. At first, the ripple will decrease each time T_A is decreased, but once the optimal value is passed, the ripple will increase as we continue to decrease T_A. Therefore, once the peak ripple starts to increase, we should decrease the size of the increment and begin *increasing* instead of decreasing T_A. Once peak ripple again stops decreasing and starts increasing, we again decrease the increment and reverse the direction. Eventually, T_A should converge to the optimum value. A slightly more sophisticated strategy for finding the optimum value of T_A is provided by the so-called *golden section search* (Press et al. 1986). This method is based on the fact that the minimum of a function $f(x)$ is known to be "bracketed" by a triplet of points $a < b < c$ provided that $f(b) < f(a)$ and $f(b) < f(c)$. Once an initial bracket is established, the span of the bracket can be methodically decreased until the three points a, b, and c converge on the abscissa of the minimum. The name "golden section" comes from the fact that the most efficient search results when the middle point of the bracket is a fraction distance 0.61803 from one endpoint and 0.38197 from the other. A C function **goldenSearch( )**, provided in Listing 12.3, performs a golden section search for our specific application. This function calls **fsDesign( ), cgdFirResponse( ), normalizeResponse( ), findSbPeak( )**, and **setTrans( )**. All of these have been discussed previously, with the exception of **setTrans( )**, which is provided in Listing 12.4. For the single-sample case this function is extremely simple, but we shall maintain it as a separate function to facilitate anticipated extensions for the case of multiple samples in the transition band that will be treated in Secs. 12.5 and 12.6. The inputs accepted by **goldenSearch** are as follows:

firType: 1 for N odd, $h[n]$ symmetric; 2 for N even, $h[n]$ symmetric; 3 for N odd, $h[n]$ antisymmetric; 4 for N even, $h[n]$ antisymmetric

numTaps: The number of taps in the desired FIR filter

Hd[ ]: The positive-frequency samples of the desired magnitude response

tol: The tolerance used to terminate the golden section search

numFreqPts: The number of samples in the (one-sided) discrete-frequency approximation to the filter's continuous-frequency response

bandConfig[ ]: An array containing filter configuration information as explained above for **findSbPeak( )**

The function provides two outputs—the peak stop-band value of the magnitude response is provided as the function's return value, and the corresponding abscissa (frequency) is written into ***fmin**.

Example 12.4 For a 21-tap lowpass filter, find the value for the transition-band sample $H_d[5]$ such that the peak stop-band ripple is minimized.

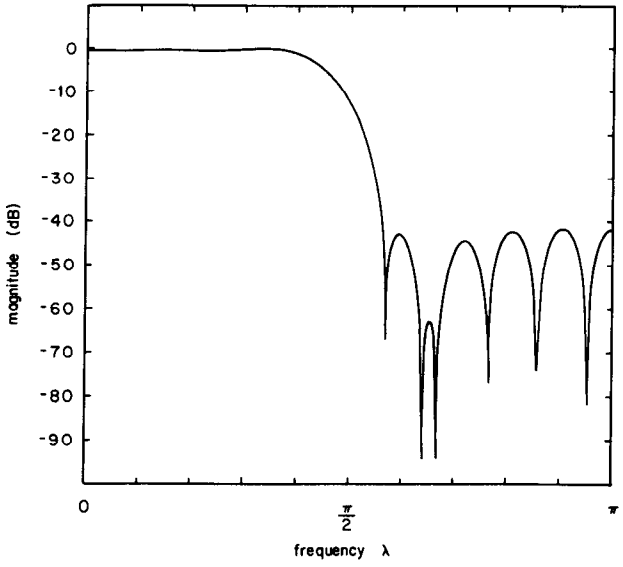

Figure 12.9 Magnitude response of 21-tap filter from Example 12.4.

solution The optimal value for $H_d[5]$ is 0.400147, and the corresponding amplitude response is shown in Fig. 12.9. The filter coefficients are listed in Table 12.5. Compared to the case where $H_d[5] = 0.5$, the peak stop-band ripple has been reduced by 11.2 dB.

12.5 Optimization with Two Transition-Band Samples

The optimization problem gets a bit more difficult when there are two or more samples in the transition band. Let's walk through the case of a type 1

TABLE 12.5 Coefficients for the Filter of Example 12.4

$h[0] = h[20] =$	0.009532
$h[1] = h[19] =$	0.002454
$h[2] = h[18] =$	−0.018536
$h[3] = h[17] =$	−0.018963
$h[4] = h[16] =$	0.025209
$h[5] = h[15] =$	0.044232
$h[6] = h[14] =$	−0.029849
$h[7] = h[13] =$	−0.094246
$h[8] = h[12] =$	0.032593
$h[9] = h[11] =$	0.314324
$h[10] =$	0.466498

lowpass filter with 21 taps having a desired response specified by

$$H_d[n] = \begin{cases} 1.0 & 0 \le |n| \le 4 \\ H_B & |n| = 5 \\ H_A & |n| = 6 \\ 0.0 & 7 \le |n| \le 10 \end{cases}$$

The values of H_A and H_B will be optimized to produce the filter having the smallest peak stop-band ripple.

1. Letting $H_B = 1$ and using a stopping tolerance of 0.01 in the single-sample **goldenSearch( )** function from Sec. 12.4, we find that the peak stop-band ripple is minimized for $H_A = 0.398227$. Thus we have defined one point in the H_A-H_B plane; specifically ($H_{A1} = 0.398227$, $H_{B1} = 1.0$).
2. We define a second point in the plane by setting $H_B = 0.97$ and once again searching for the optimum H_A value that minimizes the peak stop-band ripple. This yields a second point at (0.376941, 0.97).
3. The two points (0.398227, 1) and (0.376941, 0.97) can then be used to define a line in the H_A-H_B plane as shown in Fig. 12.10. Our ultimate goal is to determine the ordered pair (H_A, H_B) that minimizes the peak stop-band ripple of the filter. In the vicinity of (H_{A1}, 1), the line shown in Fig. 12.10 is the "best" path along which to search and is therefore called the *line of steepest descent*. On the way to achieving our ultimate goal, a useful intermediate goal is to find the point along the line at which the filter's stop-band ripple is minimized. In order to use the single-sample search procedure from Sec. 12.4 to search along this line, we can define positions on the line in terms of their projections onto the H_A axis. To evaluate the filter response for a given value of H_A, we need to have H_B expressed as a function of H_A. The slope of the line is easily determined from points 1 and 2 as

$$m = \frac{1 - 0.97}{0.398227 - 0.376941} = 1.4093$$

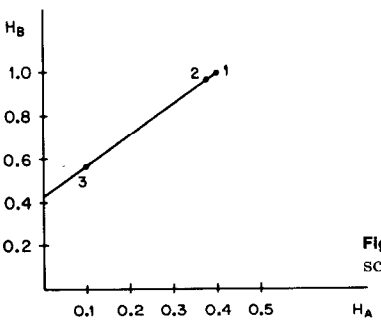

Figure 12.10 Line of steepest descent plotted in the H_A-H_B plane.

Thus we can write

$$H_B = 1.4093H_A + b \tag{12.4}$$

where b is the H_B intercept. We can then solve for b by substituting the values for H_A, H_B at point 1 into (12.4) to obtain

$$\begin{aligned} b &= H_B - 1.4093H_A \\ &= 1 - 1.4093(0.398227) = 0.438779 \end{aligned}$$

Thus the line of steepest descent is defined in the H_A-H_B plane as

$$H_B = 1.4093H_A + 0.438779 \tag{12.5}$$

The nature of the filter design problem requires that $0 \le H_A \le 1$ and $0 \le H_B \le 1$. Furthermore, examination of (12.5) indicates that $H_B < H_A$ for all values of H_A between zero and unity. Thus, the fact that H_B must not exceed unity can be used to further restrict the values of H_A. We find that $H_B = 1$ for $H_A = 0.39823$. Therefore, the search along the line is limited to values of H_A such that $0 \le H_A \le 0.39823$. The point along the line (12.5) at which the peak stop-band ripple is minimized is found to be (0.099248, 0.57863). The peak stop-band ripple at this point is -66.47 dB.

4. The ripple performance of -66.47 is respectable, but it is not the best that we can do. The straight line shown in Fig. 12.10 is in fact just an extrapolation from points 1 and 2. Generally, the actual *path* of steepest descent will not be a straight line and will diverge farther from the extrapolated line as the distance from point 1 increases. Thus when we find the optimum point (labeled as point 3) *lying along the straight line*, we really have not found the optimum point *in general*. One way to deal with this situation is to hold H_B constant at the value corresponding to point 3 and then find the optimal value of H_A—without constraining H_A to lie on the line. This results in point 4 as shown in Fig. 12.11. (Figure 12.11 uses a different scale than does Fig. 12.10 so that fine details can be more clearly shown.) The coordinates of point 4 are (0.98301, 0.57863).

5. We now perturb H_B by taking 97 percent of the value corresponding to point 4 [that is, $H_B = (0.97)(0.57863) = 0.561271$]. Searching for the value of H_A that minimizes the peak stop-band ripple, we obtain point 5 at (0.085145, 0.561271).

6. The two points (0.099248, 0.57863) and (0.085145, 0.561271) can then be used to define the new line of steepest descent shown in Fig. 12.11. Using the approach discussed above in 3, we then find the point along the line at which the peak stop-band ripple is minimized. This point is found to be (0.098592, 0.579014), and the corresponding peak ripple is -69.680885 dB.

7. We can continue this process of defining lines of steepest descent and optimizing along the line until the change in peak stop-band ripple from one iteration to the next is smaller than some preset limit. Typically, the opti-

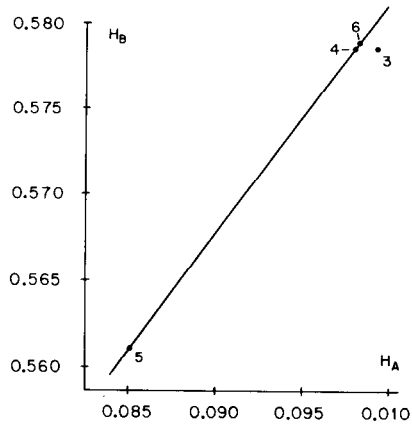

Figure 12.11 Second line of steepest descent.

mization is terminated when the peak ripple changes by less than 0.1 dB between iterations. Using this criterion, the present design converges after the fourth line of steepest descent is searched to find the point ($H_A = 0.098403$, $H_B = 0.579376$) where the peak stop-band ripple is -71.08 dB.

Programming considerations

Optimizing the value of H_A, with H_B expressed as a function of H_A, requires some changes to the way in which the function **findSbPeak( )** interfaces to the function **goldenSearch( )**. In the single-sample-transition case, the search was conducted with H_A as the independent variable supplied (in the appropriate location of **Hd[ ]**) to **findSbPeak( )**. For the two-sample-transition case, the software has been designed to conduct the search in terms of the displacement ρ measured along an arbitrary line. (This approach is more general than it needs to be for the two-sample case, but doing things this way makes extension to three or more samples relatively easy—see Sec. 12.6 for details.) The function **findSbPeak( )** "expects" to have the H_A and H_B values "plugged into" the appropriate locations in the array **Hd[ ]**. The function **goldenSearch2( )** given in Listing 12.5 has been modified to include a call to **setTransition( )** before each call to **findSbPeak( )**. The function **setTransition( )**, shown in Listing 12.6, accepts ρ as an input and resolves it into the H_A and H_B components needed by **findSbPeak( )** for computation of the impulse response and the subsequent estimation of the continuous-frequency amplitude response. The line along which ρ is being measured is specified to **setTransition( )** via the **origins[ ]** and **slopes[ ]** arrays. The values of H_A and H_B corresponding to $\rho = 0$ are passed in **origins[1]** and **origins[2]**, respectively. The changes in H_A and H_B corresponding to $\Delta\rho = 1$ are passed in **slopes[1]** and **slopes[2]**, respectively. Setting **slopes[1]** $= 1$ and **origins[1]** $= 0$ is the correct way to specify $H_A = \rho$. (Note that if we set **slopes[1]** $= 1$, **origins[1]** $= 0$, **slopes[2]** $= 0$ and **origins[2]** $= 0$, the single-sample case can be handled as a special case of the two-sample case, since

these values are equivalent to setting $H_A = \rho$ and $H_B = 0$.) The iterations of the optimization strategy are mechanized by the function **optimize2( )** given in Listing 12.7. After each call to **goldenSearch2( )**, the function **optimize2( )** uses the function **dumpRectComps( )** (shown in Listing 12.8) to print the H_A and H_B projections of the value returned by **goldenSearch2( )**.

Example 12.5 Complete the design of the 21-tap filter that was started at the beginning of this section.

solution As mentioned previously, when **goldenSearch2( )** is used with a stopping tolerance of 0.01, the example design converges after four lines of steepest descent have been searched. Each line involves 3 points—2 points to define the line plus 1 point at which the ripple is minimized. The coordinates and peak stop-band ripple levels for the 12 points of the example design are listed in Table 12.6. Each of these points required 8 iterations of **goldenSearch2( )**. The impulse response coefficients for the filter corresponding to the transition-band values of $H_A = 0.098403$ and $H_B = 0.579376$ are listed in Table 12.7. The corresponding magnitude response is plotted in Fig. 12.12.

TABLE 12.6 Points Generated in the Optimization Procedure for Example 12.5

Iteration	H_A	H_B	Stop-band peak, dB
1	0.398227	1.0	−42.22
2	0.376941	0.97	−42.76
3	0.099248	0.578630	−66.47
4	0.098301	0.578630	−69.93
5	0.085145	0.561271	−65.87
6	0.098592	0.579014	−69.68
7	0.098301	0.579014	−71.05
8	0.085145	0.561643	−65.20
9	0.098473	0.579241	−70.89
10	0.098301	0.579241	−71.02
11	0.085145	0.561864	−64.61
12	0.098403	0.579376	−71.08

TABLE 12.7 Impulse Response Coefficients for the Filter of Example 12.5

$h[0] = h[20]$	= 0.002798
$h[1] = h[19]$	= 0.004783
$h[2] = h[18]$	= −0.006541
$h[3] = h[17]$	= −0.018285
$h[4] = h[16]$	= 0.007862
$h[5] = h[15]$	= 0.042175
$h[6] = h[14]$	= −0.007896
$h[7] = h[13]$	= −0.092308
$h[8] = h[12]$	= 0.007530
$h[9] = h[11]$	= 0.313553
$h[10]$	= 0.492659

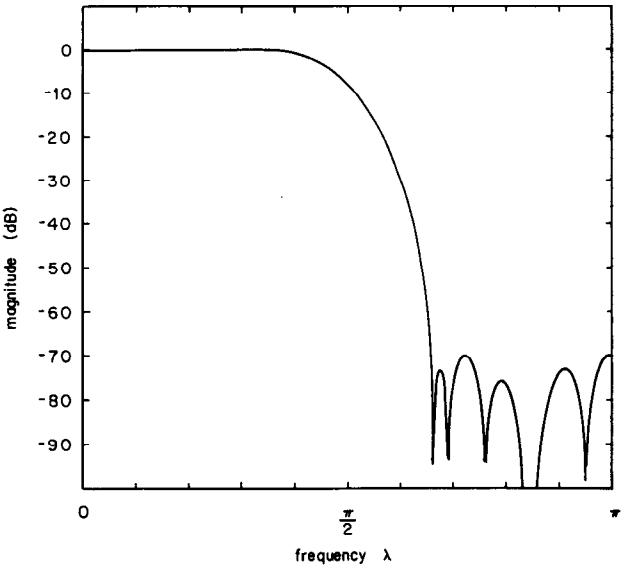

Figure 12.12 Magnitude response for Example 12.5.

Careful examination of the values in Table 12.6 reveals several anomalies. Points 1, 2, 4, 5, 7, 8, 10, and 11 define lines of steepest descent; and points 3, 6, 9, and 12 are the corresponding optimal points along these lines. The ripple performance of the "optimal" point 6 is −69.68 while the performance at point 4 is −69.93. These two points lie on the same line, and the performance at point 4 is better than the performance at point 6. A similar situation occurs with points 7 and 9. Such behavior indicates that the stopping criterion for **goldenSearch2( )** is not stringent enough, thereby allowing the search to stop before the best point on the line is found.

Example 12.6 Redesign the filter of Example 12.5 using **tol** = 0.001 instead of **tol** = 0.01.

solution The number of iterations required for each point increases from 8 to 14, but the design procedure terminates after only two lines of steepest descent. The coordinates and peak stop-band ripple levels for the six points of this design are listed in Table 12.8. The impulse response coefficients are listed in Table 12.9.

TABLE 12.8 Points Generated in the Optimization Procedure for Example 12.6

Iteration	H_A	H_B	Stop-band peak, dB
1	0.399133	1.0	−42.24
2	0.377674	0.97	−42.73
3	0.100240	0.582148	−70.46
4	0.100220	0.582148	−70.34
5	0.087517	0.564683	−65.10
6	0.100425	0.582429	−70.39

TABLE 12.9 Impulse Response Coefficients for the Filter of Example 12.6

$h[0] = h[20]$	$= 0.002636$
$h[1] = h[19]$	$= 0.004775$
$h[2] = h[18]$	$= -0.006170$
$h[3] = h[17]$	$= -0.018170$
$h[4] = h[16]$	$= 0.007275$
$h[5] = h[15]$	$= 0.042024$
$h[6] = h[14]$	$= -0.007122$
$h[7] = h[13]$	$= -0.092186$
$h[8] = h[12]$	$= 0.006629$
$h[9] = h[11]$	$= 0.313507$
$h[10]$	$= 0.493605$

Comparison of Tables 12.6 and 12.8 reveals that performance obtained in Example 12.6 is 0.7 dB worse than the performance obtained in Example 12.5. Furthermore, within Example 12.6, the performance at point 3 is slightly better than the performance at point 6. Possible strategies for combatting these numeric effects would be to use a "tweaking factor" larger than 97 percent, or to have the tweaking factor approach unity with successive iterations.

12.6 Optimization with Three Transition-Band Samples

Just as the two-transition-sample case was more complicated than the single-sample case, the three-sample case is significantly more complicated than the two-sample case. Let's consider the case of a type 1 lowpass filter having a desired response as shown in Fig. 12.13. (The following discussion assumes that the three variables H_A, H_B, and H_C are each assigned to one of the axes in a three-dimensional rectilinear coordinate system.)

1. Consider points along the line defined by $H_C = 1, H_B = 1$. (Note: $H_C = 1$ defines a plane parallel to the H_A-H_B plane, and $H_B = 1$ defines a plane that intersects the $H_C = 1$ plane in a line which is parallel to the H_A axis.) Use a single-variable search strategy (such as the golden section search) to locate the point along this line for which the peak stop-band ripple is minimized. Denote the value of H_A at this point as H_{A1}.
2. Consider points along the line defined by $H_C = 1, H_B = 1 - \epsilon$. Use a single-variable search strategy to locate the point along this line for which the peak stop-band ripple is minimized. Denote the value of H_A at this point as H_{A2}.
3. The points $(H_{A1}, 1)$ and $(H_{A2}, 1 - \epsilon)$ define a line in the H_A-H_B plane as shown in Fig. 12.10 for the two-sample case. [Actually the points and the line

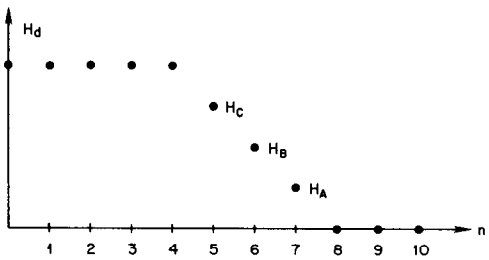

Figure 12.13 Desired response for a 21-tap type 1 filter with three samples in the transition band.

are in the plane defined by $H_C = 1$, and their *projections* onto the H_A-H_B plane are shown by Fig. 12.10. However, since the planes are parallel, everything looks the same regardless of whether we plot the points in the $H_C = 1$ plane or their projections in the H_A-H_B (that is, $H_C = 0$) plane.] In the vicinity of $(H_{A1}, 1)$, this line is the "best" path along which to search and is therefore called the *line of steepest descent.* Search along line to find the point at which the peak stop-band ripple is minimized. Denote the values of H_A and H_B at this point as H_{A3} and H_{B3}, respectively. As noted previously, the true path of steepest descent is in fact curved, and the straight line just searched is merely an extrapolation based on the two points $(H_{A1}, 1)$ and $(H_{A2}, 1 - \epsilon)$. Thus the point (H_{A3}, H_{B3}) is not a true minimum. However, this point can be taken as a starting point for a second round of steps 1, 2, and 3 which will yield a refined estimate of the minimum's location. This refined estimate can in turn be used as a starting point for a third round of steps 1, 2, and 3. This cycle of steps 1, 2, and 3 is repeated until the peak ripple at (H_{A3}, H_{B3}) changes by less than some predetermined amount (say, 0.1 dB).

Listing 12.1 fsDesign( )

```
/***********************************/
/*                                 */
/*   Listing 12.1                  */
/*                                 */
/*   fsDesign()                    */
/*                                 */
/***********************************/

int fsDesign(    int N,
                 int firType,
                 real A[],
                 real h[])
{
int n,k, status;
real x, M;

M = (N-1.0)/2.0;
status = 0;
switch (firType) {
    case 1:
        if(N%2) {
            for(n=0; n<N; n++) {
                h[n] = A[0];
                x = TWO_PI * (n-M)/N;
                for(k=1; k<=M; k++) {
                    h[n] = h[n] + 2.0*A[k]*cos(x*k);
                    }
                h[n] = h[n]/N;
                }
            }
        else
            {status = 1;}
        break;
    /*-------------------------------------*/
    case 2:
        if(N%2)
            {status = 2;}
        else {
            for(n=0; n<N; n++) {
                h[n] = A[0];
                x = TWO_PI * (n-M)/N;
            for(k=1; k<=(N/2-1); k++) {
                h[n] = h[n] + 2.0*A[k]*cos(x*k);
                }
            h[n] = h[n]/N;
            }
```

```
            }
        break;
    /*---------------------------------------*/
    case 3:
        if(N%2) {
            for(n=0; n<N; n++) {
                h[n] = 0;
                x = TWO_PI * (M-n)/N;
                for(k=1; k<=M; k++) {
                    h[n] = h[n] + 2.0*A[k]*sin(x*k);
                    }
                h[n] = h[n]/N;
                }
            }
        else
            {status = 3;}
        break;
    /*---------------------------------------*/
    case 4:
        if(N%2)
            {status = 4;}
        else {
            for(n=0; n<N; n++) {
                h[n] = A[N/2]*sin(PI*(M-n));
                x = TWO_PI * (n-M)/N;
                for(k=1; k<=(N/2-1); k++) {
                    h[n] = h[n] + 2.0*A[k]*sin(x*k);
                    }
                h[n] = h[n]/N;
                }
            }
        break;
    }
return(status);
}
```

Listing 12.2 findSbPeak( )

```
/**********************************/
/*                                */
/*   Listing 12.2                 */
/*                                */
/*   findSbPeak()                 */
/*                                */
/**********************************/


real findSbPeak( int bandConfig[],
                 int numPts,
                 real H[])
{
real peak;
int n, nBeg, nEnd, indexOfPeak;
int filterType;

filterType=bandConfig[0];

switch (filterType) {
    case 1:              /* lowpass */
        nBeg = 2*numPts*bandConfig[2]/bandConfig[5];
        nEnd = numPts-1;
        break;
    case 2:              /* highpass */
    case 3:              /* bandpass */
        nBeg = 0;
        nEnd = 2*numPts*bandConfig[1]/bandConfig[5];
        break;
    case 4:              /* bandstop */
        nBeg = 2*numPts*bandConfig[2]/bandConfig[5];
        nEnd = 2*numPts*bandConfig[3]/bandConfig[5];
        break;
    }
peak = -9999.0;
for(n=nBeg; n<nEnd; n++) {
    if(H[n]>peak) {
        peak=H[n];
        indexOfPeak = n;
        }
    }
if(filterType == 4) {     /* bandpass has second stopband */
    nBeg = 2*numPts*bandConfig[4]/bandConfig[5];
    nEnd = numPts;
    for(n=nBeg; n<nEnd; n++) {
        if(H[n]>peak) {
```

```
            peak=H[n];
            indexOfPeak = n;
            }
        }
    }
return(peak);
}
```

Listing 12.3 goldenSearch( )

```
/**********************************/
/*                                */
/*   Listing 12.3                 */
/*                                */
/*   goldenSearch()               */
/*                                */
/**********************************/


real goldenSearch(    int firType,
                      int numbTaps,
                      real Hd[],
                      real tol,
                      int numFreqPts,
                      int bandConfig[],
                      real *fmin)
{
real x0, x1, x2, x3, xmin, f0, f1, f2, f3, oldXmin;
real leftOrd, rightOrd, midOrd, midAbsc, x, xb;
real delta;
static real hh[100], H[610];
int n;
logical dbScale;
FILE *logPtr;

printf("in goldenSearch\n");
logPtr = fopen("search.log","w");

dbScale = TRUE;
/*-----------------------------------------*/
setTrans( bandConfig, 0, Hd);
fsDesign( numbTaps, firType, Hd, hh);
cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
normalizeResponse(dbScale,numFreqPts,H);
leftOrd = findSbPeak(bandConfig,numFreqPts,H);
printf("leftOrd = %f\n",leftOrd);
```

```
setTrans( bandConfig, 1.0, Hd);
fsDesign( numbTaps, firType, Hd, hh);
cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
normalizeResponse(dbScale,numFreqPts,H);
rightOrd = findSbPeak(bandConfig,numFreqPts,H);
printf("rightOrd = %f\n",rightOrd);
pause(pauseEnabled);

if(leftOrd < rightOrd) {
    midAbsc=1.0;
    for(;;) {
        printf("checkpoint 3\n");
        midAbsc = GOLD3 * midAbsc;
        setTrans( bandConfig, midAbsc, Hd);
        fsDesign( numbTaps, firType, Hd, hh);
        cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
        normalizeResponse(dbScale,numFreqPts,H);
        midOrd = findSbPeak(bandConfig,numFreqPts,H);
        printf("midOrd = %f\n",midOrd);
        if(midOrd < leftOrd) break;
        }
    }
else {
    x = 1.0;
    for(;;) {
        x = GOLD3 * x;
        midAbsc = 1.0 - x;
        printf("checkpoint 4\n");
        setTrans( bandConfig, midAbsc, Hd);
        fsDesign( numbTaps, firType, Hd, hh);
        cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
        normalizeResponse(dbScale,numFreqPts,H);
        midOrd = findSbPeak(bandConfig,numFreqPts,H);
        printf("midOrd = %f\n",midOrd);
        if(midOrd < rightOrd) break;
        }
    }
xb = midAbsc;
/*-----------------------------------------*/
x0 = 0.0;
x3 = 1.0;
x1 = xb;
x2 = xb + GOLD3 * (1.0 - xb);
printf("x0= %f, x1= %f, x2= %f, x3= %f\n",x0,x1,x2,x3);

setTrans( bandConfig, x1, Hd);
fsDesign( numbTaps, firType, Hd, hh);
cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
```

```
normalizeResponse(dbScale,numFreqPts,H);
f1 = findSbPeak(bandConfig,numFreqPts,H);

setTrans( bandConfig, x2, Hd);
fsDesign( numbTaps, firType, Hd, hh);
cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
normalizeResponse(dbScale,numFreqPts,H);
f2 = findSbPeak(bandConfig,numFreqPts,H);

oldXmin = 0.0;

for(n=1; n<=100; n++) {
    if(f1<=f2) {
        x3 = x2;
        x2 = x1;
        x1 = GOLD6 * x2 + GOLD3 * x0;
        f3 = f2;
        f2 = f1;
        setTrans( bandConfig, x1, Hd);
        fsDesign( numbTaps, firType, Hd, hh);
        cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
        normalizeResponse(dbScale,numFreqPts,H);
        f1 = findSbPeak(bandConfig,numFreqPts,H);
        printf("x0= %f, x1= %f, x2= %f, x3= %f\n",x0,x1,x2,x3);
        }
    else {
        x0 = x1;
        x1 = x2;
        x2 = GOLD6 * x1 + GOLD3 * x3;
        f0 = f1;
        f1 = f2;
        setTrans( bandConfig, x2, Hd);
        fsDesign( numbTaps, firType, Hd, hh);
        cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
        normalizeResponse(dbScale,numFreqPts,H);
        f2 = findSbPeak(bandConfig,numFreqPts,H);
        printf("x0= %f, x1= %f, x2= %f, x3= %f\n",x0,x1,x2,x3);
        }

    delta = fabs(x3 - x0);
    oldXmin = xmin;
    printf("at iter %d, delta = %f\n",n,delta);
    printf("tol = %f\n",tol);
    if(delta <= tol) break;
    }
if(f1<f2)
    {xmin = x1;
    *fmin=f1;}
```

```
else
    {xmin = x2;
    *fmin=f2;}
printf("minimum of %f at x = %f\n", *fmin, xmin);
fprintf(logFptr,"minimum of %f at x = %f\n", *fmin, xmin);
return(xmin);
}
```

Listing 12.4 setTrans( )

```
/**********************************/
/*                                */
/*   Listing 12.4                 */
/*                                */
/*   setTrans()                   */
/*                                */
/**********************************/


void setTrans(    int bandConfig[],
                  real x,
                  real Hd[])
{
int n1, n2, n3, n4;

n1 = bandConfig[1];
n2 = bandConfig[2];
n3 = bandConfig[3];
n4 = bandConfig[4];

switch (bandConfig[0]) {
    case 1:                    /* lowpass */
            Hd[n2-1] = x;
        break;
    case 2:                    /* highpass */
            Hd[n1+1] = x;
        break;
    case 3:                    /* bandpass */
            Hd[n1+1] = x;
            Hd[n4-1] = Hd[n1+1];
        break;
    case 4:                    /* bandstop */
            Hd[n2-1] = x;
            Hd[n3+1] = Hd[n2-1];
        break;
    }
return;
}
```

Listing 12.5 goldenSearch2( )

```
/***********************************/
/*                                 */
/*   Listing 12.5                  */
/*                                 */
/*   goldenSearch2()               */
/*                                 */
/***********************************/


real goldenSearch2(  real rhoMin,
                     real rhoMax,
                     int firType,
                     int numbTaps,
                     real Hd[],
                     real tol,
                     int numFreqPts,
                     real origins[],
                     real slopes[],
                     int bandConfig[],
                     real *fmin)
{
real x0, x1, x2, x3, xmin, f0, f1, f2, f3, oldXmin;
real leftOrd, rightOrd, midOrd, midAbsc, x, xb;
real delta;
static real hh[100], H[610];
int n;
logical dbScale;

dbScale = TRUE;

/*-----------------------------------------*/
setTransition( origins, slopes, bandConfig, 0, Hd);
fsDesign( numbTaps, firType, Hd, hh);
cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
normalizeResponse(dbScale,numFreqPts,H);
leftOrd = findSbPeak(bandConfig,numFreqPts,H);

setTransition( origins, slopes, bandConfig, rhoMax, Hd);
fsDesign( numbTaps, firType, Hd, hh);
cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
normalizeResponse(dbScale,numFreqPts,H);
rightOrd = findSbPeak(bandConfig,numFreqPts,H);

if(leftOrd < rightOrd) {
    midAbsc=rhoMax;
    for(;;) {
        midAbsc = GOLD3 * midAbsc;
```

```
        setTransition( origins, slopes, bandConfig, midAbsc, Hd);
        fsDesign( numbTaps, firType, Hd, hh);
        cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
        normalizeResponse(dbScale,numFreqPts,H);
        midOrd = findSbPeak(bandConfig,numFreqPts,H);
        if(midOrd < leftOrd) break;
        }
    }
else {
    x = rhoMax;
    for(;;) {
        x = GOLD3 * x;
        midAbsc = rhoMax - x;
        setTransition( origins, slopes, bandConfig, midAbsc, Hd);
        fsDesign( numbTaps, firType, Hd, hh);
        cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
        normalizeResponse(dbScale,numFreqPts,H);
        midOrd = findSbPeak(bandConfig,numFreqPts,H);
        if(midOrd < rightOrd) break;
        }
    }
xb = midAbsc;

/*-------------------------------------------*/
x0 = rhoMin;
x3 = rhoMax;
x1 = xb;
x2 = xb + GOLD3 * (rhoMax - xb);

setTransition( origins, slopes, bandConfig, x1, Hd);
fsDesign( numbTaps, firType, Hd, hh);
cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
normalizeResponse(dbScale,numFreqPts,H);
f1 = findSbPeak(bandConfig,numFreqPts,H);

setTransition( origins, slopes, bandConfig, x2, Hd);
fsDesign( numbTaps, firType, Hd, hh);
cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
normalizeResponse(dbScale,numFreqPts,H);
f2 = findSbPeak(bandConfig,numFreqPts,H);

oldXmin = 0.0;

for(n=1; n<=100; n++) {
    if(f1<=f2) {
        x3 = x2;
        x2 = x1;
        x1 = GOLD6 * x2 + GOLD3 * x0;
```

```
f3 = f2;
f2 = f1;
setTransition( origins, slopes, bandConfig, x1, Hd);
fsDesign( numbTaps, firType, Hd, hh);
cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
normalizeResponse(dbScale,numFreqPts,H);
f1 = findSbPeak(bandConfig,numFreqPts,H);
}
    else {
        x0 = x1;
        x1 = x2;
        x2 = GOLD6 * x1 + GOLD3 * x3;
        f0 = f1;
        f1 = f2;
        setTransition( origins, slopes, bandConfig, x2, Hd);
        fsDesign( numbTaps, firType, Hd, hh);
        cgdFirResponse(firType,numbTaps, hh, dbScale, numFreqPts,H);
        normalizeResponse(dbScale,numFreqPts,H);
        f2 = findSbPeak(bandConfig,numFreqPts,H);
        }

    delta = fabs(x3 - x0);
    oldXmin = xmin;
    if(delta <= tol) break;
    }
if(f1<f2)
    {xmin = x1;
    *fmin=f1;}
else
    {xmin = x2;
    *fmin=f2;}
return(xmin);
}
```

Listing 12.6 setTransition( )

```
/**********************************/
/*                                */
/*   Listing 12.6                 */
/*                                */
/*   setTransition()              */
/*                                */
/**********************************/


void setTransition(   real origins[],
                      real slopes[],
                      int bandConfig[],
```

```
                    real x,
                    real Hd[])
{
int n, nnn, n1, n2, n3, n4;

nnn = bandConfig[2] - bandConfig[1] - 1;
n1 = bandConfig[1];
n2 = bandConfig[2];
n3 = bandConfig[3];
n4 = bandConfig[4];

switch (bandConfig[0]) {
    case 1:                 /* lowpass */
        for( n=1; n<=nnn; n++) {
            Hd[n2-n] = origins[n] + x * slopes[n];
            }
        break;
    case 2:                 /* highpass */
        for( n=1; n<=nnn; n++){
            Hd[n1+n] = origins[n] + x * slopes[n];
            }
        break;
    case 3:                 /* bandpass */
        for( n=1; n<=nnn; n++) {
            Hd[n1+n] = origins[n] + x * slopes[n];
            Hd[n4-n] = Hd[n1+n];
            }
        break;
    case 4:                 /* bandstop */
        for( n=1; n<=nnn; n++) {
            Hd[n2-n] = origins[n] + x * slopes[n];
            Hd[n3+n] = Hd[n2-n];
            }
        break;
    }
return;
}
```

Listing 12.7 optimize2( )

```
/***********************************/
/*                                 */
/*   Listing 12.7                  */
/*                                 */
/*   optimize2()                   */
/*                                 */
/***********************************/


void optimize2(  real yBase,
                 int firType,
                 int numbTaps,
                 real Hd[],
                 real gsTol,
                 int numFreqPts,
                 int bandConfig[],
                 real tweakFactor,
                 real rectComps[])
{
real r1, r2, r3, x1, x2, x3, y3, minFuncVal;
real slopes[5], origins[5];
real oldMin, xMax;
for(;;)
    {
    /*-------------------------------------------------*/
    /*  do starting point for new steepest descent line  */
    slopes[1] = 1.0;
    slopes[2] = 0.0;
    origins[1] = 0.0;
    origins[2] = yBase;

    x1 = goldenSearch2(  0.0, 1.0,
                         firType,numbTaps,Hd,gsTol,numFreqPts,
                         origins,slopes,bandConfig,&minFuncVal);

/*-------------------------------------*/
/*  do perturbed point to get          */
/*    slope for steepest descent line  */

origins[2]=yBase * tweakFactor;

x2 = goldenSearch2(  0.0, 1.0, firType,numbTaps,Hd,
                     gsTol,numFreqPts,origins,slopes,
                     bandConfig,&minFuncVal);

/*-------------------------------------*/
/* define line of steepest descent     */
```

```
    /*  and find optimal point along line  */

    slopes[2] = yBase*(1-tweakFactor)/(x1-x2);
    origins[2] = yBase - slopes[2] * x1;
    xMax = (1.0 - origins[2])/slopes[2];

    x3 = goldenSearch2(  0.0, xMax, firType,numbTaps,Hd,
                         gsTol,numFreqPts,
                         origins,slopes,bandConfig,&minFuncVal);
    y3=origins[2] + x3 * slopes[2];
    /*-------------------------------------------------------------*/
    /*  if ripple at best point on current line is within specified  */
    /*    tolerance of ripple at best point on previous line,        */
    /*    then stop; otherwise stay in loop and define a new line    */
    /*    starting at the best point on line just completed.         */

    if(abs(oldMin-minFuncVal)<0.01) break;
    oldMin = minFuncVal;
    yBase = y3;
    }
rectComps[0] = x3;
rectComps[1] = origins[2] + x3 * slopes[2];
return;
}
```

Listing 12.8 dumpRectComps( )

```
/**********************************/
/*                                */
/*   Listing 12.8                 */
/*                                */
/*   dumpRectComps()              */
/*                                */
/**********************************/


void dumpRectComps(  real origins[],
                     real slopes[],
                     int numTransSamps,
                     real x)
{
real rectComp;
int n;

for(n=0; n<numTransSamps; n++)
    {
    rectComp = origins[n+1] + x * slopes[n+1];
    printf("rectComp[%d] = %f\n",n,rectComp);
    }
return;
}
```