Chapter

14

IIR Filters

The general form for an *infinite impulse response* (IIR) filter's output $y[k]$ at time k is given by

$$y[n] = \sum_{n=1}^{N} a_n y[k-n] + \sum_{m=0}^{M} b_m x[k-m] \tag{14.1}$$

This equation indicates that the filter's output is a linear combination of the present input, the M previous inputs, and the N previous outputs. The corresponding system function is given by

$$H(z) = \frac{\sum_{m=0}^{M} b_m z^{-m}}{1 - \sum_{n=1}^{N} a_n z^{-n}} \tag{14.2}$$

where at least one of the a_n is nonzero and at least one of the roots of the denominator is not exactly cancelled by one of the roots of the numerator. For a stable filter, all the poles of $H(z)$ must lie inside the unit circle, but the zeros can lie anywhere in the z plane. It is usual for M, the number of zeros, to be less than or equal to N, the number of poles. Whenever the number of zeros exceeds the number of poles, the filter can be separated into an FIR filter with $M - N$ taps in cascade with an IIR filter with N poles and N zeros. Therefore, IIR design techniques are conventionally restricted to cases for which $M \leq N$.

Except for the special case in which all poles lie on the unit circle (in the z plane), it is not possible to design an IIR filter having exactly linear phase. Therefore, unlike FIR design procedures that are concerned almost exclusively with the magnitude response, IIR design procedures are concerned with both the magnitude response and phase response.

14.1 Frequency Response of IIR Filters

The frequency response of an IIR filter can be computed from the coefficients a_n and b_m as

$$H[k] = \frac{\sum_{m=0}^{L-1} \beta_m \exp(j2\pi mk/L)}{\sum_{n=0}^{L-1} \alpha_n \exp(j2\pi k/L)} \tag{14.3}$$

where $\alpha_n = \begin{cases} 1 & n = 0 \\ -a_n & 0 < n \le N \\ 0 & N < n \end{cases}$

$$\beta_m = \begin{cases} b_m & 0 \le m \le M \\ 0 & M < m \end{cases}$$

A C function that uses (14.3) to compute the response for an IIR filter is provided in Listing 14.1.

14.2 IIR Realizations

A direct realization of Eq. (14.1) is shown in Fig. 14.1 using the signal flow graph notation introduced in Sec. 4.4. The structure shown is known as the *direct form I realization* or *direct form I structure* for the IIR system represented by (14.1). Examination of the figure reveals that the system can be viewed as two systems in cascade—the first system using $x[k-M]$ through $x[k]$ to generate an intermediate signal that we will call $w[k]$ and the second

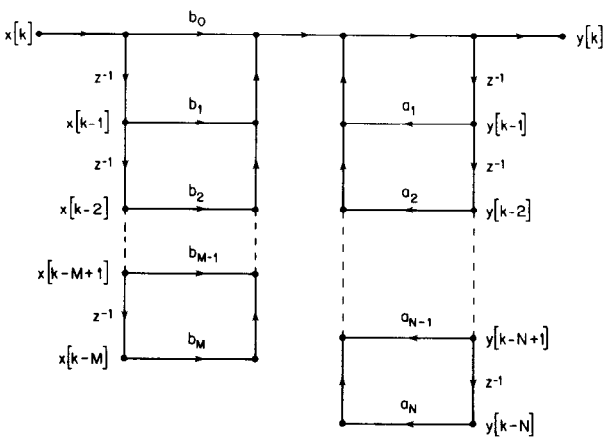

Figure 14.1 Signal flow graph of direct form I realization for an IIR system.

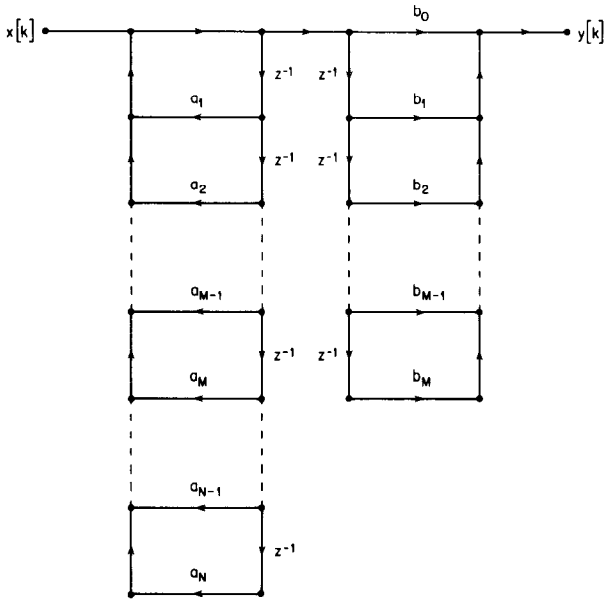

Figure 14.2 Signal flow graph of Fig. 14.1 with cascade order reversed.

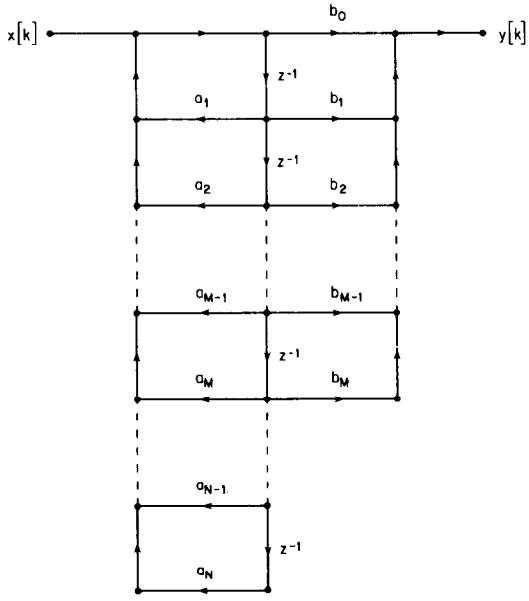

Figure 14.3 Signal flow graph of direct form II realization for an IIR system.

system using $w[k]$ and $y[k-N]$ through $y[k-1]$ to generate $y[k]$. Since these two systems are LTI systems, the order of the cascade can be reversed to yield the equivalent system shown in Fig. 14.2. Examination of this figure reveals that the unit delays in parallel running down the center of the diagram can be paired such that within a pair the two delays each take the same input signal. This fact can be exploited to merge the two delay chains into a single chain as shown in Fig. 14.3. The structure shown in this figure is known as the *direct form II realization* of the IIR system represented by (14.1).

14.3 Impulse Invariance

The basic idea behind the impulse-invariance approach is a very simple one—the unit sample response of the digital filter is set equal to a sequence of uniformly spaced samples from the impulse response of an analog filter:

$$h[n] = h_a(nT) \tag{14.4}$$

(An analog filter used in this context is usually refered to as a "prototype" filter.) This approach is conceptually simple, but from a practical viewpoint, evaluation of (14.4) is not a straightforward matter. By definition, for an *infinite* impulse response filter, the sequence $h[n]$ will be nonzero over an infinite domain of n. Furthermore, based on the s-plane-to-z-plane mapping discussed in Sec. 9.2, we can conclude that the imposition of (14.4) will not result in a simple relationship between the frequency response corresponding to $h[n]$ and the frequency response corresponding to $h_a(t)$. In fact, this relationship can be shown to be

$$H(e^{j\lambda}) = \frac{1}{T}\sum_{k=-\infty}^{\infty} H_a\left(j\frac{\lambda + 2\pi k}{T}\right) \tag{14.5}$$

where $h[n] \overset{\text{DTFT}}{\longleftrightarrow} H(e^{j\lambda})$

$h_a(t) \overset{\text{FT}}{\longleftrightarrow} H_a(j\omega)$

Put simply, Eq. (14.5) indicates that $H(e^{j\lambda})$ will be an aliased version of $H_a(j\omega)$. The only way the aliasing can be avoided is if $H_a(j\omega)$ is band limited such that

$$H_a(j\omega) = 0 \qquad \text{for } |\omega| \geq \frac{\pi}{T} \tag{14.6}$$

If (14.6) is satisfied, then

$$H(e^{j\lambda}) = \frac{1}{T}H_a\left(j\frac{\lambda}{T}\right) \qquad |\lambda| \leq \pi \tag{14.7}$$

For a practical analog filter, Eq. (14.6) will never be satisfied exactly, but the impulse-invariance method can be used to advantage with responses that are nonzero but negligible beyond some frequency.

The transfer function of the analog prototype filter can be expressed in the form of a partial-fraction expansion as

$$H_a(s) = \sum_{k=1}^{N} \frac{A_k}{s - s_k} \tag{14.8}$$

where the s_k are the poles of $H_a(s)$ and the A_k are given by

$$A_k = [(s - s_k)H_a(s)]\big|_{s = s_k}$$

Based on transform pair 8 from Table 2.2, the impulse response can then be written as

$$h_a(t) = \sum_{k=1}^{N} A_k e^{s_k t} u(t) \tag{14.9}$$

The unit-sample response of the digital filter is then formed by sampling the prototype filter's impulse response to obtain

$$h[n] = \sum_{k=1}^{N} A_k (e^{s_k T})^n u(t) \tag{14.10}$$

The corresponding system function for the digital filter $H(z)$ is obtained as the z transform of (14.10):

$$H(z) = \sum_{k=1}^{N} \frac{A_k}{1 - e^{s_k T} z^{-1}} \tag{14.11}$$

Based on the foregoing, we can state the following algorithm for impulse-invariant design of an IIR filter.

Algorithm 14.1 Impulse-invariant design of IIR filters

Step 1. Obtain the transfer function $H_a(s)$ for the desired analog prototype filter. (The material provided in Chaps. 3 through 6 will prove useful here.)

Step 2. For $k = 1, 2, \ldots, N$, determine the poles s_k of $H_a(s)$ and compute the coefficients A_k using

$$A_k = [(s - s_k)H_a(s)]\big|_{s = s_k} \tag{14.12}$$

Step 3. Using the coefficients A_k obtained in step 2, generate the digital filter system function $H(z)$ as

$$H(z) = \sum_{k=1}^{N} \frac{A_k}{1 - \exp(s_k T) z^{-1}} \tag{14.13}$$

where T is the sampling interval of the digital filter.

Step 4. The result obtained in step 3 will be a sum of fractions. Obtain a common denominator, and express $H(z)$ as a ratio of polynomials in z^{-1} in the form

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{k=1}^{N} a_k z^{-k}} \tag{14.14}$$

Step 5. Use the a_k and b_k obtained in step 4 to realize the filter in any of the structures given in Sec. 14.1.

Example 14.1 Use the technique of impulse invariance to derive a lowpass IIR digital filter from a second-order Butterworth analog filter with a 3-dB cutoff frequency of 3 KHz. The sampling rate for the digital filter is 30,000 samples per second.

solution From Sec. 3.1 we obtain the normalized-transfer function for a second-order Butterworth filter as

$$H(s) = \frac{1}{(s - s_1)(s - s_2)}$$

where $s_1 = \cos\frac{3\pi}{4} + j \sin\frac{3\pi}{4}$

$$= \frac{-\sqrt{2}}{2} + j\frac{\sqrt{2}}{2}$$

$$s_2 = \cos\frac{5\pi}{4} + j \sin\frac{5\pi}{4}$$

$$= \cos\frac{3\pi}{4} - j \sin\frac{3\pi}{4}$$

$$= \frac{-\sqrt{2}}{2} - j\frac{\sqrt{2}}{2}$$

The specified cutoff frequency of $f = 3000$ yields $\omega_c = 6000\pi$, and the denormalized response (see Sec. 2.9) is given by

$$H_a(s) = \frac{\omega_c^2}{(s - \omega_c s_1)(s - \omega_c s_2)}$$

$$= \frac{\omega_c^2}{[s + \omega_c(\sqrt{2}/2) - j\omega_c(\sqrt{2}/2)][s + \omega_c(\sqrt{2}/2) + j\omega_c(\sqrt{2}/2)]}$$

The partial-fraction expansion of $H_a(s)$ is given by

$$H_a(s) = \frac{A_1}{s + \omega_c(\sqrt{2}/2) - j\omega_c(\sqrt{2}/2)} + \frac{A_2}{s + \omega_c(\sqrt{2}/2) + j\omega_c(\sqrt{2}/2)}$$

where $A_1 = \frac{-j\sqrt{2}}{2\omega_c}$

$$A_2 = \frac{j\sqrt{2}}{2\omega_c}$$

Using these values for A_1 and A_2 plus the fact that

$$\omega_c T = \frac{6000\pi}{30{,}000} = \frac{\pi}{5}$$

we obtain from Eq. (14.13) the discrete system function $H(z)$ as

$$H(z) = \frac{-j\sqrt{2}/(2\omega_c)}{1 - \exp\left(\frac{-\pi\sqrt{2}}{10} + j\frac{\pi\sqrt{2}}{10}\right)z^{-1}} + \frac{j\sqrt{2}/(2\omega_c)}{1 - \exp\left(\frac{-\pi\sqrt{2}}{10} - j\frac{\pi\sqrt{2}}{10}\right)z^{-1}}$$

$$= \frac{2.06797 \times 10^{-5} z^{-1}}{1 - 1.158045z^{-1} + 0.4112407z^{-2}}$$

Programming considerations

Step 1. Butterworth, Chebyshev, and Bessel filters are "all-pole" filters—their transfer functions have no finite zeros. Closed-form expressions are available for the poles of Butterworth [Eq. (3.2)] and Chebyshev [Eq. (4.4)] filters. The poles of Bessel filters can be readily obtained by finding the roots of the denominator polynomial as discussed in Chap. 6. The transfer function for an elliptical filter has both poles and zeros. The poles are readily available by using the quadratic formula to find the denominator roots for each factor in Eq. (5.22). The zeros $\pm j\alpha\sqrt{a_i}$ are obtained by inspection of Eq. (5.22). The software for performing the impulse-invariance transformation is therefore designed to accept $H_a(s)$ specified as an array of poles and an array of zeros.

Step 2. Evaluation of A_k for step 2 of the algorithm is straightforward. The coefficients A_k can be written as $A_k = N_{Ak}/D_{Ak}$ where the numerator N_{Ak} is obtained as

$$N_{Ak} = \begin{cases} H_0 \prod_{m=1}^{M} (p_k - q_m) & M \neq 0 \\ H_0 & M = 0 \end{cases}$$

and q_m is the mth zero of $H_a(s)$, p_k is the kth pole of $H_a(s)$, and M is the total number of zeros. Equation (14.12) can be evaluated using simple arithmetic—there is no symbolic manipulation needed. The denominator D_{Ak} is obtained as

$$D_{Ak} = \prod_{\substack{n=1 \\ n \neq k}}^{N} (p_k - p_n)$$

Step 3. Evaluation of $H(z)$ is more than plain, straightforward arithmetic. At this point, for each value of k, the coefficient A_k is known and the coefficient $\exp(s_k T)$ can be evaluated. However, z remains a variable and hence will demand some special consideration. To simplify the notation in

the subsequent development, let us rewrite $H(z)$ as

$$H(z) = \sum_{k=1}^{N} \frac{A_k}{1 + \beta_k z^{-1}} \tag{14.15}$$

where $\beta_k = -\exp(s_k T)$

Step 4. For the summation in (14.15), the common denominator will be the product of each summand's denominator:

$$D(z) = \prod_{k=1}^{N} (1 + \beta_k z^{-1}) \tag{14.16}$$

To see how (14.16) can be easily evaluated by computer, let's examine the sequence of partial products $\{D_k(z)\}$ encountered in the evaluation:

$$D_1(z) = (1 + \beta_1 z^{-1})$$

$$D_2(z) = (1 + \beta_2 z^{-1})\, D_1(z) = D_1(z) + \beta_2 z^{-1}\, D_1(z)$$

$$D_3(z) = (1 + \beta_3 z^{-1})\, D_2(z) = D_2(z) + \beta_3 z^{-1}\, D_2(z)$$

$$D_4(z) = (1 + \beta_4 z^{-1})\, D_3(z) = D_3(z) + \beta_4 z^{-1}\, D_3(z)$$

$$\vdots$$

$$D(z) = D_N(z) = (1 + \beta_N z^{-1})\, D_{N-1}(z) = D_{N-1}(z) + \beta_N z^{-1}\, D_{N-1}(z)$$

Examination of this sequence reveals that the partial product $D_k(z)$ at iteration k can be expressed in terms of the partial product $D_{k-1}(z)$ as

$$D_k(z) = D_{k-1}(z) + \beta_k z^{-1}\, D_{k-1}(z)$$

The partial product $D_{k-1}(z)$ will be a $(k-1)$-degree polynomial in z^{-1}:

$$D_{k-1}(z) = \delta_0 (z^{-1})^0 + \delta_1 (z^{-1})^1 + \delta_2 (z^{-1})^2 + \cdots + \delta_{k-1} (z^{-1})^{k-1}$$

The product $\beta_k z^{-1}\, D_{k-1}(z)$ is then given by

$$\beta_k z^{-1} D_{k-1}(z) = \delta_0 \beta_k (z^{-1})^1 + \delta_1 \beta_k (z^{-1})^2 + \delta_2 \beta_k (z^{-1})^3 + \cdots + \delta_{k-1} \beta_k (z^{-1})^k$$

and $D_k(z)$ is given by

$$D_k(z) = \delta_0 (z^{-1})^0 + (\delta_1 + \delta_0 \beta_k)(z^{-1})^1 + (\delta_2 + \delta_1 \beta_k)(z^{-1})^2 + \cdots + (\delta_{k-1} + \delta_{k-2} \beta_k)(z^{-1})^{k-1} + \delta_{k-1} \beta_k (z^{-1})^k$$

Therefore, we can conclude that if δ_n is the coefficient for the $(z^{-1})^n$ term in $D_{k-1}(z)$, then the coefficient for the $(z^{-1})^n$ term in $D_k(z)$ is $(\delta_n + \delta_{n-1} \beta_k)$ with the proviso that $\delta_k \triangleq 0$ in $D_{k-1}(z)$. The polynomial $D_{k-1}(z)$ can be represented in the computer as an array of k coefficients, with the array index

corresponding to the subscript on δ and the superscript (exponent) on (z^{-1}): **delta[0]** $= \delta_0$, **delta[1]** $= \delta_1$, and so on. The coefficients for the partial product $D_k(z)$ can be obtained from the coefficients for $D_{k-1}(z)$ as indicated by the following fragment of pseudocode:

```
for( j=k; j >= 1; j--)
    {delta[j] = delta[j] + beta * delta[j-1];}
```

The loop is executed in reverse order so that the coefficients can be updated "in place" without prematurely overwriting the old values. Notice that I referred to the fragment shown above as "pseudocode." In actuality, both **delta[ ]** and **beta** are complex valued; and the arithmetic operations shown in the fragment are incorrect. The following code fragment performs the complex arithmetic correctly, but all the complex functions tend to obscure the algorithm that is more clearly conveyed by the pseudocode above:

```
for( j=k; j >= 1; j--)
    {delta[j] = cAdd(delta[j], cMult(beta, delta[j-1]));}
```

If this fragment is placed within an outer loop with k ranging from 1 to **numPoles**, the final values in **delta[n]** will be the coefficients a_n for Eq. (14.14).

For the summation in Eq. (14.13), the numerator can be computed as

$$N(z) = \sum_{k=1}^{N} \left[A_k \prod_{\substack{n=1 \\ n \neq k}}^{N} (1 - \beta_n z^{-1}) \right] \tag{14.17}$$

For each value of k, the product in (14.17) can be evaluated in a manner similar to the way in which the denominator is evaluated. The major difference is that the factor $(1 - \beta_k z^{-1})$ is not included in the product. It is then a simple matter to add the coefficients of each of the N products to obtain the coefficients for the numerator polynomial $N(z)$. A complete function for computing the coefficients a_n and b_n is provided in Listing 14.2.

14.4 Step Invariance

One major drawback to filters designed via the impulse-invariance method is their sensitivity to the specific characteristics of the input signal. The digital filter's unit-sample response is a sampled version of the prototype filter's impulse response. However, the prototype filter's response to an arbitrary input cannot in general be sampled to obtain the digital filter's response to a sampled version of the same arbitrary input. In many applications a filter's step response is of more concern than is the filter's impulse response. In such cases, the impulse-invariance technique can be modified to design a digital filter based on the principle of step invariance.

Algorithm 14.2 Step-invariant design of IIR filters

Step 1. Obtain the transfer function $H_a(s)$ for the desired analog prototype filter.

Step 2. Multiply $H_a(s)$ by $1/s$ to obtain $G_a(s)$, the filter's response to the unit step function.

Step 3. For $k = 1, 2, \ldots, N$, determine the poles s_k of $G_a(s)$ and compute the coefficients A_k using

$$A_k = [(s - s_k)G_a(s)]\big|_{s=s_k}$$

Step 4. Using the coefficients A_k obtained in step 3, generate the system function $G(z)$ as

$$G(z) = \sum_{k=1}^{N} \frac{A_k}{1 - \exp(s_k T)z^{-1}}$$

Step 5. Multiply $G(z)$ by $(1 - z^{-1})$ to remove the z transform of a unit step and thereby obtain $H(z)$ as

$$H(z) = (1 - z^{-1}) \sum_{k=1}^{N} \frac{A_k}{1 - \exp(s_k T)z^{-1}}$$

Step 6. Obtain a common denominator for the terms in the summation of step 5, and express $H(z)$ as a ratio of polynomials in z^{-1} in the form

$$G(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 - \sum_{k=1}^{N} a_k z^{-k}}$$

Step 7. Use the a_k and b_k obtained in step 6 to realize the filter in any of the structures given in Sec. 14.1.

Programming considerations

The step-invariance method is similar to the impulse-invariance method, with two important differences. In step 2 of Algorithm 14.2, the transfer function $H_a(s)$ is multiplied by $1/s$. Assuming that $H_a(s)$ is represented in terms of its poles and zeros, multiplication by $1/s$ is accomplished by simply adding a pole at $s = 0$. (Strictly speaking, if the analog filter has a zero at $s = 0$, multiplication by $1/s$ creates a pole at $s = 0$, which cancels the zero. However, since none of the analog prototype filters within the scope of this book have zeros at $s = 0$, we shall construct the software without provisions for handling a zero at $s = 0$.)

In step 5 of Algorithm 14.2, the system function $G(z)$ is multiplied by $(1 - z^{-1})$ to remove the z transform of a unit step and thereby obtain the

system function $H(z)$. Conceptually, this multiplication is appropriately located in step 5. However, for ease of implementation it makes sense to defer the multiplication until after the coefficients a_n and b_n are generated in step 6. A function modified to perform the step-invariance technique is provided in Listing 14.3.

Listing 14.1 iirResponse( )

```
/***********************************/
/*                                 */
/*   Listing 14.1                  */
/*                                 */
/*   iirResponse()                 */
/*                                 */
/***********************************/


void iirResponse(struct complex a[],
                 int bigN,
                 struct complex b[],
                 int bigM,
                 int numberOfPoints,
                 logical dbScale,
                 real magnitude[],
                 real phase[])
{
static struct complex response[MAXPOINTS];
int k, n, m;
real sumRe, sumIm, phi;

/*-------------------------------------------------*/
/*  compute DFT of H(z) numerator                  */

for( m=0; m<numberOfPoints; m++) {
    sumRe = 0.0;
    sumIm = 0.0;
    printf("\r%d  @@@",m);
    for(n=0; n<=bigM; n++) {
        printf("\b\b\b%3d",n);
        phi = 2.0 * PI * m * n / (2.0*numberOfPoints);
        printf("b[%d] = (%e, %e)\n",n,b[n].Re,b[n].Im);
        sumRe += b[n].Re * cos(phi) + b[n].Im * sin(phi);
        sumIm += b[n].Im * cos(phi) - b[n].Re * sin(phi);
        }
    response[m] = cmplx(sumRe, sumIm);
    printf("response = (%e, %e)\n",response[m].Re, response[m].Im);
    }

/*-------------------------------------------------*/
/*  compute DFT of H(z) denominator                */

for( m=0; m<numberOfPoints; m++) {
    sumRe = 1.0;
    sumIm = 0.0;
```

```
    for(n=1; n<=bigN; n++) {
        phi = 2.0 * PI * m * n / (2.0*numberOfPoints);
        sumRe += -a[n].Re * cos(phi) - a[n].Im * sin(phi);
        sumIm += -a[n].Im * cos(phi) + a[n].Re * sin(phi);
        }
    response[m] = cDiv(response[m],cmplx(sumRe, sumIm));
    }
/*-------------------------------------------------*/
/*  compute magnitude and phase of response        */

for( m=0; m<numberOfPoints; m++) {
    phase[m] = arg(response[m]);
    if(dbScale)
        {magnitude[m] = 20.0 * log10(cAbs(response[m]));}
    else
        {magnitude[m] = cAbs(response[m]);}
    printf("mag = %e\n",magnitude[m]);
    }
return;
}
```

Listing 14.2 impulseInvar( )

```
/**********************************/
/*                                */
/*   Listing 14.2                 */
/*                                */
/*   impulseInvar()               */
/*                                */
/**********************************/


void impulseInvar(   struct complex pole[],
                     int numPoles,
                     struct complex zero[],
                     int numZeros,
                     real hZero,
                     real bigT,
                     struct complex a[],
                     struct complex b[])
{
int k, n, j, maxCoef;
struct complex delta[MAXPOLES];
struct complex bigA[MAXPOLES];
struct complex beta, denom, numer, work2;
```

```
for(j=0; j<MAXPOLES; j++) {
    delta[j] = cmplx(0.0,0.0);
    a[j] = cmplx(0.0,0.0);
    b[j] = cmplx(0.0,0.0);
    }
/*-------------------------------------------------*/
/*  compute partial fraction expansion coefficients  */
for( k=1; k<=numPoles; k++) {
    numer = cmplx(hZero,0.0);
    for(n=1; n<=numZeros; n++)
        { numer = cMult(numer, cSub(pole[n], zero[n]));}
    denom = cmplx(1.0,0.0);
    for( n=1; n<=numPoles; n++) {
        if(n==k) continue;
        denom = cMult(denom, cSub(pole[k],pole[n]));
        }
    bigA[k] = cDiv(numer,denom);
    }
/*-------------------------------------*/
/*  compute numerator coefficients     */
for( k=1; k<=numPoles; k++) {
    delta[0] = cmplx(1.0, 0.0);
    for(n=1; n<MAXPOLES; n++)
        {delta[n] = cmplx(0.0,0.0);}
    maxCoef = 0;
    for( n=1; n<=numPoles; n++) {
        if(n==k) continue;
        maxCoef++;
        beta = sMult(-1.0, cExp(sMult(bigT,pole[n])));
        for(j=maxCoef; j>=1; j--)
            { delta[j] = cAdd( delta[j], cMult( beta, delta[j-1]));}
        }
    for( j=0; j<numPoles; j++)
        { b[j] = cAdd(b[j], cMult( bigA[k], delta[j])); }
    }
/*-------------------------------------*/
/*  compute denominator deltaficients   */
a[0] = cmplx(1.0,0.0);
for( n=1; n<=numPoles; n++) {
    beta = sMult(-1.0, cExp(sMult(bigT,pole[n])));
    for( j=n; j>=1; j--)
        { a[j] = cAdd( a[j], cMult( beta, a[j-1]));}
    }
for( j=1; j<=numPoles; j++)
    { a[j] = sMult(-1.0,a[j]);}
return;
}
```

Listing 14.3 stepInvar( )

```
/**********************************/
/*                                */
/*   Listing 14.3                 */
/*                                */
/*   stepInvar()                  */
/*                                */
/**********************************/

void stepInvar(  struct complex pole[],
                 int numPoles,
                 struct complex zero[],
                 int numZeros,
                 real hZero,
                 real bigT,
                 struct complex a[],
                 struct complex b[])
{
int k, n, j, maxCoef;
struct complex delta[MAXPOLES];
struct complex bigA[MAXPOLES];
struct complex beta, denom, numer, work2;

for(j=0; j<MAXPOLES; j++) {
    delta[j] = cmplx(0.0,0.0);
    a[j] = cmplx(0.0,0.0);
    b[j] = cmplx(0.0,0.0);
    }
pole[0] = cmplx(0.0,0.0);
/*-------------------------------------------------*/
/*  compute partial fraction expansion coefficients  */
for( k=0; k<=numPoles; k++) {
    numer = cmplx(hZero,0.0);
    for(n=1; n<=numZeros; n++)
        { numer = cMult(numer, cSub(pole[n], zero[n]));}
    denom = cmplx(1.0,0.0);
    for( n=0; n<=numPoles; n++) {
        if(n==k) continue;
        denom = cMult(denom, cSub(pole[k],pole[n]));
        }
    bigA[k] = cDiv(numer,denom);
    }
/*-------------------------------------*/
/*  compute numerator coefficients     */
for( k=1; k<=numPoles; k++) {
    delta[0] = cmplx(1.0, 0.0);
    for(n=1; n<MAXPOLES; n++)
```

```
        {delta[n] = cmplx(0.0,0.0);}
    maxCoef = 0;
    for( n=0; n<=numPoles; n++) {
        if(n==k) continue;
        maxCoef++;
        beta = sMult(-1.0, cExp(sMult(bigT,pole[n])));
        for(j=maxCoef; j>=1; j--)
            { delta[j] = cAdd( delta[j], cMult( beta, delta[j-1]));}
        }
    for( j=0; j<numPoles; j++)
        { b[j] = cAdd(b[j], cMult( bigA[k], delta[j])); }

/* multiply by 1-z**(-1)   */
    beta = cmplx(-1.0,0.0);
    for(j=numPoles+1; j>=1; j--) {
        b[j] = cAdd(b[j], cMult(beta, b[j-1]));}

    }
/*-------------------------------------*/
/*  compute denominator coefficients   */
a[0] = cmplx(1.0,0.0);
for( n=1; n<=numPoles; n++) {
    beta = sMult(-1.0, cExp(sMult(bigT,pole[n])));
    for( j=n; j>=1; j--)
        { a[j] = cAdd( a[j], cMult( beta, a[j-1]));}
    }
for( j=1; j<=numPoles; j++)
    { a[j] = sMult(-1.0,a[j]);}
return;
}
```