Chapter

5

Elliptical Filters

By allowing ripples in the pass band, Chebyshev filters obtain better selectivity than Butterworth filters do. Elliptical filters improve upon the performance of Chebyshev filters by permitting ripples in *both* the pass band and stop band. The response of an elliptical filter satisfies

$$|H(j\omega)|^2 = \frac{1}{1 + \epsilon^2 R_n^2(\omega, L)}$$

where $R_n(\omega, L)$ is an nth-order *Chebyshev rational function* with ripple parameter L. Elliptical filters are sometimes called *Cauer filters.*

5.1 Parameter Specification

As shown in Chap. 3, determination of the (amplitude-normalized) transfer function for a Butterworth lowpass filter requires specification of just two parameters—cutoff frequency ω_c and filter order n. Determination of the transfer function for a Chebyshev filter requires specification of these two parameters plus a third—pass-band ripple (or stop-band ripple for inverse Chebyshev). Determination of the transfer function for an elliptical filter requires specification of the filter order n plus the following four parameters, which are depicted in Fig. 5.1:

A_p = maximum pass-band loss, dB

A_s = minimum stop-band loss, dB

ω_p = pass-band cutoff frequency

ω_s = stop-band cutoff frequency

The design procedures presented in this chapter assume that the maximum pass-band amplitude is unity. Therefore, A_p is the size of the pass-band

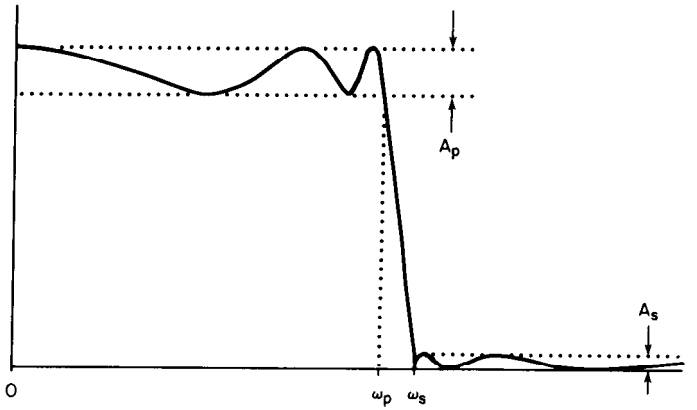

Figure 5.1 Frequency response showing parameters used to specify an elliptical filter.

ripples, and A_s is the size of the stop-band ripples. Any four of the five filter parameters can be specified independently, with the fifth then being fixed by the nature of the elliptical filter's response. The usual design strategy involves specifying A_p, A_s, ω_p, and ω_s based upon requirements of the intended application. Algorithm 5.1, as follows, can then be used to compute the minimum value of n for which an elliptical filter can yield the desired performance. Since n must be an integer, not all combinations of A_p, A_s, ω_p, and ω_s can be realized exactly. The design procedure presented in this chapter can yield a filter that meets the specified A_p, A_s, and ω_p and that meets *or exceeds* the specification on A_s.

Algorithm 5.1 Determining the required order for elliptical filters

Step 1. Based upon requirements of the intended application, determine the maximum stop-band loss A_p and minimum stop-band loss A_s in decibels.

Step 2. Based on requirements of the intended application, determine the pass-band cutoff frequency ω_p and stop-band cutoff frequency ω_s.

Step 3. Using ω_p and ω_s, compute *selectivity factor* k as $k = \omega_p/\omega_s$.

Step 4. Using the selectivity factor computed in step 3, compute the *modular constant* q using

$$q = u + 2u^5 + 15u^9 + 150u^{13} \tag{5.1}$$

where $u = \dfrac{1 - \sqrt[4]{1-k^2}}{2(1 + \sqrt[4]{1-k^2})}$ (5.2)

Step 5. Using the values of A_p and A_s, determined in step 1, compute the *discrimination factor* D as

$$D = \frac{10^{A_s/10} - 1}{10^{A_p/10} - 1} \tag{5.3}$$

Step 6. Using the value of D from step 5 and the value of q from step 4, compute the minimum required order n as

$$n = \left\lceil \frac{\log 16D}{\log(1/q)} \right\rceil \tag{5.4}$$

where $\lceil x \rceil$ denotes the smallest integer equal to or greater than x.

The actual minimum stop-band loss provided by any given combination of A_p, ω_p, ω_s, and n is given by

$$A_s = 10 \log\left(1 + \frac{10^{A_p/10} - 1}{16q^n}\right) \tag{5.5}$$

where q is the modular constant given by Eq. (5.1).

Example 5.1 Use Algorithm 5.1 to determine the minimum order for an elliptical filter for which $A_p = 1$, $A_s \geq 50.0$, $\omega_p = 3000.0$, and $\omega_s = 3200.0$.

solution

$$k = \frac{3000}{3200} = 0.9375$$

$$u = 0.12897$$

$$q = 0.12904$$

$$D = \frac{10^5 - 1}{10^{0.01} - 1} = 4{,}293{,}093.82$$

$$n = \lceil 8.81267 \rceil = 9$$

A C function **cauerOrderEstim( )**, which implements Algorithm 5.1, is provided in Listing 5.1. This function also computes the actual minimum stop-band loss in accordance with Eq. (5.5).

5.2 Normalized-Transfer Function

The design of elliptical filters is greatly simplified by designing a frequency-normalized filter having the appropriate response characteristics, and then frequency-scaling this design to the desired operating frequency. The simplification comes about because of the particular type of normalizing that is performed. Instead of normalizing so that either a 3-dB bandwidth or the ripple bandwidth equals unity, an elliptical filter is normalized so that

$$\sqrt{\omega_{pN}\omega_{sN}} = 1 \tag{5.6}$$

where ω_{pN} and ω_{sN} are, respectively, the normalized pass-band cutoff frequency and the normalized stop-band cutoff frequency. If we let α represent the frequency-scaling factor such that

$$\omega_{pN} = \frac{\omega_p}{\alpha} \qquad \omega_{sN} = \frac{\omega_s}{\alpha} \tag{5.7}$$

then we can solve for the value of α by substituting (5.7) into (5.6) to obtain

$$\sqrt{\frac{\omega_p \omega_s}{\alpha^2}} = 1$$

$$\alpha = \sqrt{\omega_p \omega_s} \tag{5.8}$$

As it turns out, the only way that the frequencies ω_{pN} and ω_{sN} enter into the design procedure (given by Algorithm 5.2) is via the selectivity factor k that is given by

$$k = \frac{\omega_{pN}}{\omega_{sN}} = \frac{\omega_p/\alpha}{\omega_s/\alpha} = \frac{\omega_p}{\omega_s} \tag{5.9}$$

Since Eq. (5.9) indicates that k can be obtained directly from the desired ω_p and ω_s, we can design a *normalized* filter without having to determine the normalized frequencies ω_{pN} and ω_{sN}! However, once a normalized design is obtained, the frequency-scaling factor α as given by (5.8) *will* be needed to frequency-scale the design to the desired operating frequency.

Algorithm 5.2 Generating normalized-transfer functions for elliptical filters

Step 1. Use Algorithm 5.1 or any other equivalent method to determine a viable combination of values for A_p, A_s, ω_p, ω_s, and n.

Step 2. Using ω_p and ω_s, compute the *selectivity factor* k as $k = \omega_p/\omega_s$.

Step 3. Using the selectivity factor computed in step 3, compute the *modular constant* q using

$$q = u + 2u^5 + 15u^9 + 150u^{13} \tag{5.10}$$

where $u = \dfrac{1 - \sqrt[4]{1-k^2}}{2(1 + \sqrt[4]{1-k^2})}$ (5.11)

Step 4. Using the values of A_p and n from step 1, compute V as

$$V = \frac{1}{2n} \ln\left(\frac{10^{A_p/20} + 1}{10^{A_p/20} - 1}\right) \tag{5.12}$$

Step 5. Using the value of q from step 3 and the value of V from step 4, compute p_0 as

$$p_0 = \left| \frac{q^{1/4} \sum_{m=0}^{\infty} (-1)^m q^{m(m+1)} \sinh[(2m+1)V]}{0.5 + \sum_{m=1}^{\infty} (-1)^m q^{m^2} \cosh 2mV} \right| \tag{5.13}$$

Step 6. Using the value of k from step 2 and the value of p_0 from step 5, compute W as

$$W = \left[\left(1 + \frac{p_0^2}{k}\right)(1 + kp_0^2) \right]^{1/2} \tag{5.14}$$

Step 7. Determine r, the number of quadratic sections in the filter, as $r = n/2$ for even n, and $r = (n-1)/2$ for odd n.

Step 8. For $i = 1, 2, \ldots, r$, compute X_i as

$$X_i = \frac{2q^{1/4} \sum_{m=0}^{\infty} (-1)^m q^{m(m+1)} \sin[(2m+1)\mu\pi/n]}{1 + 2 \sum_{m=1}^{\infty} (-1)^m q^{m^2} \cos(2m\mu\pi/n)} \tag{5.15}$$

where $\mu = \begin{cases} i & n \text{ odd} \\ i - \frac{1}{2} & n \text{ even} \end{cases}$

Step 9. For $i = 1, 2, \ldots, r$, compute Y_i as

$$Y_i \left[\left(1 - \frac{X_i^2}{k}\right)(1 - kX_i^2) \right]^{1/2} \tag{5.16}$$

Step 10. For $i = 1, 2, \ldots, r$, use the W, X_i, and Y_i from steps 6, 8, and 9; compute the coefficients a_i, b_i, and c_i as

$$a_i = \frac{1}{X_i^2} \tag{5.17}$$

$$b_i = \frac{2p_0 Y_i}{1 + p_0^2 X_i^2} \tag{5.18}$$

$$c_i = \frac{(p_0 Y_i)^2 + (X_i W)^2}{(1 + p_0^2 X_i^2)^2} \tag{5.19}$$

Step 11. Using a_i and c_i, compute H_0 as

$$H_0 = \begin{cases} p_0 \prod_{i=1}^{r} \dfrac{c_i}{a_i} & n \text{ odd} \\ 10^{-A_p/20} \prod_{i=1}^{r} \dfrac{c_i}{a_i} & n \text{ even} \end{cases} \tag{5.20}$$

Step 12. Finally, compute the normalized transfer function $H_N(s)$ as

$$H_N(s) = \frac{H_0}{d} \prod_{i=1}^{r} \frac{s^2 + a_i}{s^2 + b_i s + c_i} \tag{5.21}$$

where $d = \begin{cases} s + p_0 & n \text{ odd} \\ 1 & n \text{ even} \end{cases}$

A C function **cauerCoeffs( )**, which implements steps 1 through 11 of Algorithm 5.2, is provided in Listing 5.2. Step 12 is implemented separately in the C function **cauerFreqResponse( )** shown in Listing 5.3, since Eq. (5.21) must be reevaluated for each value of frequency.

Example 5.2 Use Algorithm 5.2 to obtain the coefficients of the normalized-transfer function for the ninth-order elliptical filter having $A_p = 0.1$ dB, $\omega_p = 3000$ rad/s, and $\omega_s = 3200$ rad/s. Determine the actual minimum stop-band loss.

solution Using the formulas from Algorithm 5.2 plus Eq. (5.5), we obtain

$$q = 0.129041 \qquad V = 0.286525 \qquad p_0 = 0.470218$$

$$W = 1.221482 \qquad r = 4 \qquad A_s = 51.665651$$

The coefficients X_i, Y_i, a_i, b_i, and c_i obtained via steps 8 through 10 for $i = 1, 2, 3, 4$ are listed in Table 5.1. Using (5.20), we obtain $H_0 = 0.015317$. The normalized-frequency response of this filter is shown in Figs. 5.2, 5.3, and 5.4. (The phase response shown in Fig. 5.4 may seem a bit peculiar. At first glance, the discontinuities in the phase response

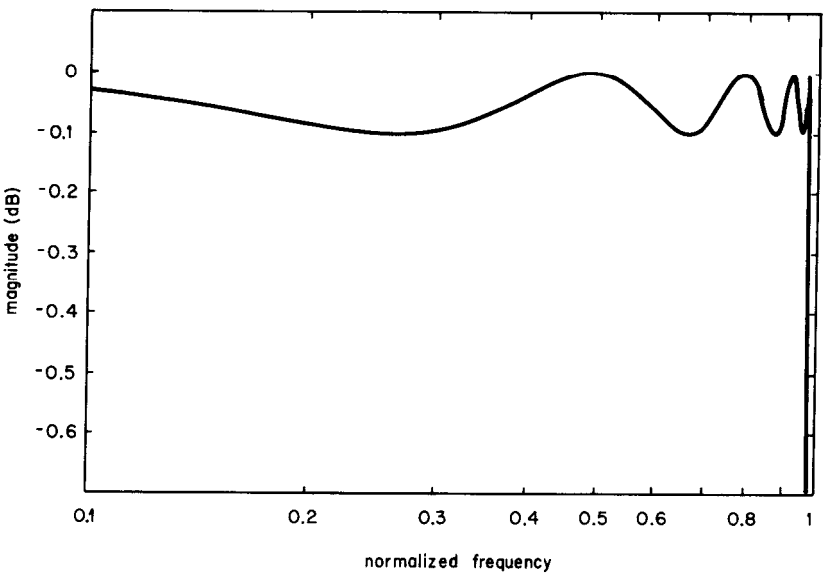

Figure 5.2 Pass-band magnitude response for Example 5.2.

TABLE 5.1 Coefficients for Example 5.2

i	X_i	Y_i	a_i	b_i	c_i
1	0.4894103	0.7598211	4.174973	0.6786235	0.4374598
2	0.7889940	0.3740371	1.606396	0.3091997	0.7415493
3	0.9196814	0.1422994	1.182293	0.1127396	0.8988261
4	0.9636668	0.0349416	1.076828	0.0272625	0.9538953

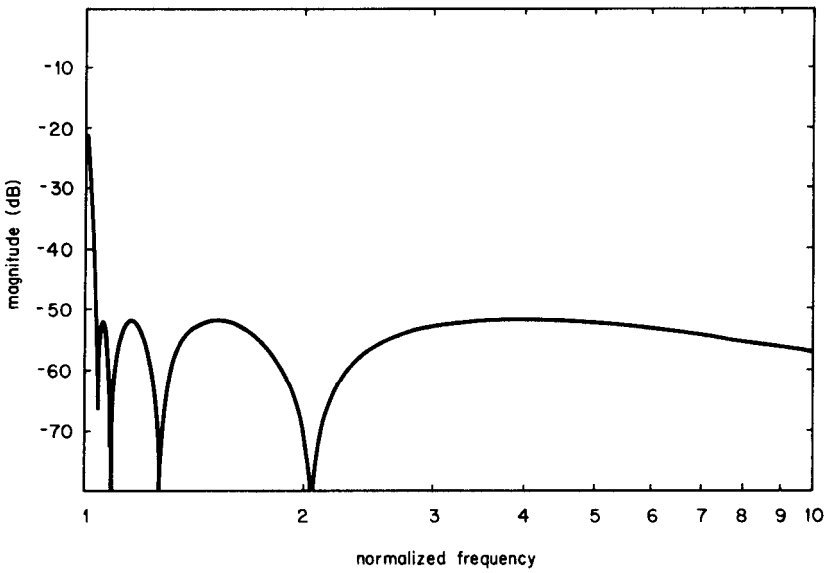

Figure 5.3 Stop-band magnitude response for Example 5.2.

might be taken for jumps of 2π caused by the $+\pi$ to $-\pi$ "wraparound" of the arctangent operation. However, this is not the case. The discontinuities in Fig. 5.4 are jumps of π that coincide with the nulls in the magnitude response.

5.3 Denormalized-Transfer Function

As noted in Sec. 2.9, if we have a response normalized for $\omega_{cN} = 1$, we can frequency-scale the transfer function to yield an identical response for $\omega_c = \alpha$ by multiplying each pole and each zero by α and dividing the overall transfer function by $\alpha^{(n_z - n_p)}$ where n_z is the number of zeros and n_p is the number of poles. An elliptical filter has a transfer function of the form given by (5.20). For odd n, there is a real pole at $s = p_0$ and r can conjugate pairs of poles that are roots of

$$s^2 + b_i s + c_i = 0 \qquad i = 1, 2, \ldots, r$$

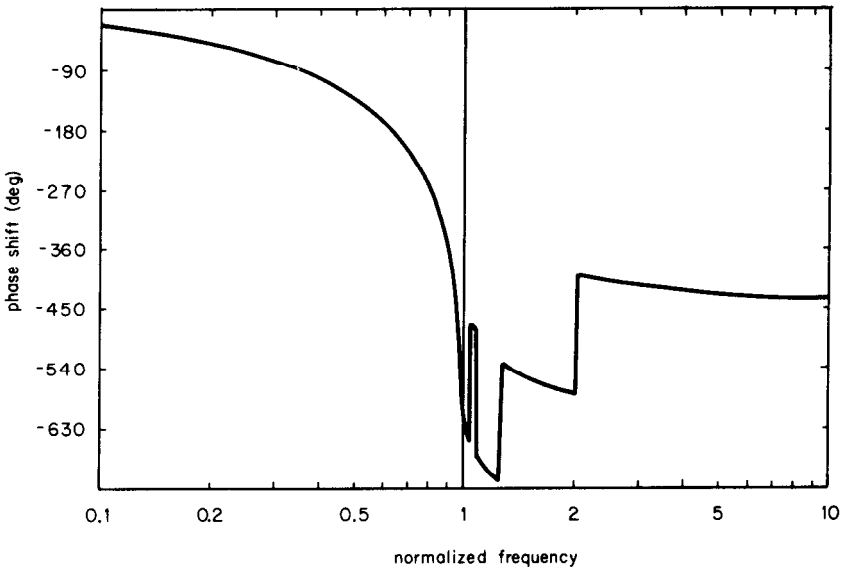

Figure 5.4 Phase response for Example 5.2.

Using the quadratic formula, the ith pair of complex pole values can be expressed as

$$p_i = \frac{-b_i \pm \sqrt{b_i^2 - 4c_i}}{2}$$

The zeros of the normalized-transfer function occur at $s = \pm j\sqrt{a_i}$, $i = 1, 2, \ldots, r$. For even n, the number of poles equals the number of zeros so $\alpha^{(n_z - n_p)} = 1$. For odd n, $n_z - n_p = -1$, so the transfer function must be divided by $1/\alpha$ or multiplied by α. If we multiply the poles and zeros by α and multiply the overall transfer function by 1 or α as appropriate, we obtain the frequency-scaled transfer function $H(s)$ as

$$H(s) = K \prod_{i=1}^{r} \frac{s^2 + \alpha^2 a_i}{s^2 + \alpha b_i s + \alpha^2 c_i} \tag{5.22}$$

where $K = \begin{cases} \dfrac{H_0 \alpha}{s + \alpha p_0} & n \text{ odd} \\ H_0 & n \text{ even} \end{cases}$

Comparison of Eqs. (5.21) and (5.22) indicates that the frequency rescaling

consists of making the following substitutions in (5.21):

$\alpha^2 a_i$ replaces a_i

$\alpha^2 c_j$ replaces c_i

αb_i replaces b_i

$H_0 \alpha$ replaces H_0 (n odd)

αp_0 replaces p_0 (n odd)

A C function **cauerRescale( )**, which makes these substitutions, is given in Listing 5.4.

Listing 5.1 cauerOrderEstim( )

```
/***********************************/
/*                                 */
/*   Listing 5.1                   */
/*                                 */
/*   cauerOrderEstim()             */
/*                                 */
/***********************************/

void cauerOrderEstim( real omegaPass,
                      real omegaStop,
                      real maxPassLoss,
                      real minStopLoss,
                      int *order,
                      real *actualMinStopLoss)
{
real k, u, q, dd, kk, lambda, w, mu, om;
real sum, term, denom, numer, sigma, v;
int i, m, r;

k=omegaPass/omegaStop;                    /* Alg. 5.1, step 3 */

kk=sqrt(sqrt(1.0 - k*k));              /* Eq (5.2) */
u=0.5*(1.0-kk)/(1.0+kk);

q = 150.0 * ipow(u,13);                   /* Eq (5.1) */
q = q + 15.0 * ipow(u,9);
q = q + 2.0 * ipow(u,5);
q = q + u;

dd = pow(10.0, minStopLoss/10.0) - 1.0;            /* Eq (5.3) */
dd = dd/ (pow(10.0,maxPassLoss/10.0) - 1.0);

*order = ceil( log10(16.0*dd) / log10(1.0/q));    /* Eq (5.4) */

                                                  /* Eq (5.5) */
numer = pow(10.0, (maxPassLoss/10.0))-1.0;
*actualMinStopLoss = 10.0 * log10(numer/(16*ipow(q,*order))+1.0);
return;
}
```

Listing 5.2 cauerCoeffs( )

```
/**********************************/
/*                                */
/*   Listing 5.2                  */
/*                                */
/*   cauerCoeffs()                */
/*                                */
/**********************************/

void cauerCoeffs(real omegaPass,
                 real omegaStop,
                 real maxPassLoss,
                 int order,
                 real aa[],
                 real bb[],
                 real cc[],
                 int *numSecs,
                 real *h2zero,
                 real *p2zero)
{
real k, kk, u, q, vv, ww, mu, xx, yy;
real sum, term, denom, numer;
int i, m, r;

k=omegaPass/omegaStop;            /* Alg 5.2, step 2 */

kk=sqrt(sqrt(1.0 - k*k));     /* Eq (5.11) */
u=0.5*(1.0-kk)/(1.0+kk);

q = 150.0 * ipow(u,13);           /* Eq (5.10) */
q = q + 15.0 * ipow(u,9);
q = q + 2.0 * ipow(u,5);
q = q + u;

                                          /* Eq (5.12) */
numer = pow(10.0,maxPassLoss/20.0)+1.0;
vv = log( numer / (pow(10.0, maxPassLoss/20.0)-1))/(2.0*order);

sum = 0.0;                    /* Eq (5.13) */
for( m=0; m<5; m++) {
    term = ipow(-1.0,m);
    term = term * ipow(q, m*(m+1));
    term = term * sinh((2*m+1) * vv);
    fprintf(dumpFile,"for m=%d, term = %e\n",m,term);
    sum = sum + term;
    }
numer = 2.0 * sum * sqrt(sqrt(q));
```

```
sum = 0.0;
for( m=1; m<5; m++) {
    term = ipow(-1.0,m);
    term = term * ipow(q,m*m);
    term = term * cosh(2.0 * m * vv);
    sum = sum + term;
    }
denom = 1.0 + 2.0*sum;
*pZero = fabs(numer/denom);

ww = 1.0 + k * *pZero * *pZero;                    /* Eq (5.14) */
ww = sqrt(ww * (1.0 + *pZero * *pZero/k));

r = (order-(order%2))/2;        /* Alg 5.2, step 7 */
*numSecs = r;

for(i=1; i<=r; i++) {     /* loop for Alg 5.2, steps 8, 9, 10 */
    if(order%2)
        {mu = i;}
    else
        {mu = i - 0.5;}
    sum = 0.0;                          /* Eq (5.15) numerator */
    for(m=0; m<5; m++) {
        term = ipow(-1.0,m);
        term = term * ipow(q, m*(m+1));
        term = term * sin( (2*m+1) * PI * mu / order);
        sum = sum + term;
        }
   numer = 2.0 * sum * sqrt(sqrt(q));

sum = 0.0;                          /* Eq (5.15) denominator */
for(m=1; m<5; m++) {
    term = ipow(-1.0,m);
    term = term * ipow(q,m*m);
    term = term * cos(2.0 * PI * m * mu / order);
    fprintf(dumpFile,"for m=%d, term = %e\n",m,term);
    sum = sum + term;
    }
denom = 1.0 + 2.0 * sum;
xx = numer/denom;

yy = 1.0 - k * xx*xx;                          /* Eq (5.16) */
yy = sqrt(yy * (1.0-(xx*xx/k)));

aa[i] = 1.0/(xx*xx);                           /* Eq (5.17) */

denom = 1.0 + ipow(*pZero*xx, 2);              /* Eq (5.18) */
bb[i] = 2.0 * *pZero * yy/denom;
```

```
    denom = ipow(denom,2);                    /* Eq (5.19) */
    numer = ipow(*pZero*yy,2) + ipow(xx*ww,2);
    cc[i] = numer/denom;
}

term = 1.0;                                   /* Eq (5.20) */
for(i=1; i<=r; i++) {
    term = term * cc[i]/aa[i];
    }
if(order%2)
    {term = term * *pZero;}
else
    {term = term * pow(10.0, maxPassLoss/(-20.0));}
*hZero = term;
return;
}
```

Listing 5.3 cauerFreqResponse( )

```
/*********************************/
/*                               */
/*   Listing 5.3                 */
/*                               */
/*   cauerFreqResponse()         */
/*                               */
/*********************************/

void cauerFreqResponse(   int order,
                          real aa[],
                          real bb[],
                          real cc[],
                          real hZero,
                          real pZero,
                          real frequency,
                          real *magnitude,
                          real *phase)
{
double normalizedFrequency;
int r, k, ix, i;
struct complex s, cProd, cTermNumer, cTermDenom;

r = (order-(order%2))/2;
s = cmplx(0.0, frequency);

if(order%2) {
    cTermDenom = cAdd(s, cmplx(pZero, 0.0));
    cProd = cDiv(cmplx(1.0,0.0), cTermDenom);
    cProd = sMult(hZero, cProd);
    }
```

```
 else {
      cProd = cmplx(h2zero,0.0);
      }
 for (i=1; i<=r; i++) {
      cTermNumer=cMult(s,s);
      cTermDenom=cAdd(cTermNumer,sMult(bb[i],s));
      cTermNumer.Re = cTermNumer.Re + aa[i];
      cTermDenom.Re = cTermDenom.Re + cc[i];
      cProd = cMult(cProd, cTermNumer);
      cProd = cDiv(cProd, cTermDenom);
      }
*magnitude = 20.0* log10(cAbs( cProd));
*phase = 180.0 * arg(cProd)/PI;
return;
}
```

Listing 5.4 cauerRescale( )

```
/***********************************/
/*                                 */
/*   Listing 5.4                   */
/*                                 */
/*   cauerRescale()                */
/*                                 */
/***********************************/

void cauerRescale(    int order,
                      real aa[],
                      real bb[],
                      real cc[],
                      real *h2zero,
                      real *p2zero,
                      real alpha)
{
real alphaSqrd;
int r, i;

alphaSqrd = alpha*alpha;

if( order%2) {
    r = (order-1)/2;
    *h2zero = *h2zero * alpha;
    *p2zero = *p2zero * alpha;
    }
```

```
else {
    r = order/2;
    }
for(i=1; i<=r; i++) {
    aa[i] = aa[i] * alphaSqrd;
    cc[i] = cc[i] * alphaSqrd;
    bb[i] = bb[i] * alpha;
    }
}
```