Mobile Digital Filter Design Toolbox

Somsak Sukittanon and Josh Potts
University of Tennessee at Martin
Department of Engineering
Martin, TN USA
ssukitta@utm.edu, joslpott@ut.utm.edu

Abstract—Mobile devices have become very useful in many fields as portable computation devices. They function as data analysis and collection tools. Though they have become common place in many fields, there are few applications in the area of Digital Signal Processing (DSP). In this paper, we have developed a filter design toolbox for mobile devices, particularly iOS devices. This toolbox allows for the design of filter of user specified order and type, export of filter coefficients via email, frequency and impulse response plotting, and real time filtered playback. The limited memory and processing power of the devices required special mathematics for the filter design to be derived.

I. INTRODUCTION

Digital signal processing (DSP) is common in today’s technical world and can be found in many applications. Audio data manipulation relies heavily on DSP in the digital age for filtering data, digitization, and analysis. Powers system also rely on DSP for applications such as better detecting AC current frequencies. Filter design, an advanced topic in DSP, is a vital tool that is used in many fields. Currently, filter design is done mostly on desktop computer platforms with programs such as Matlab or the open source alternative Octave. Both offer powerful tools for digital signal processing.

Smaller systems for DSP implementation have been developed before for microprocessors, and now mobile devices. Mobile devices are becoming the preferred computational device for many different fields, even though they lack the power of a traditional desktop computer. In Physics [1], data collecting tools such as compasses and high resolution cameras are being connected to mobile devices to analyze data. In chemistry, the TsoiChem application [2] is enhancing and reinforcing topics covered in organic chemistry. Mobile applications [3] have been developed to help students learn introductory programming concepts in the engineering field. One of the most utilized devices for these type of applications is the Apple iPad/iPhone, which run the iOS. The primary difference between the iPad and iPhone is the screen size. These devices use the Cocoa Touch programming, which is based on Objective-C language. These devices differ from their desktop counterparts in that multiple applications cannot be displayed simultaneously on the screen, and are limited to one user interface at a time. The devices are also limited to the memory available to applications, and applications cannot access anything that would require administrator privileges on a typical desktop. Mobile devices offer much more portability than desktops, but suffer from a limited screen size as a result. Though mobile device have proliferated these many fields, DSP is still a relatively untouched area on mobile devices. One such application is iDSP [4], an iPhone application that allows the user to do data filtering and analysis using graphical block diagrams. Matlab [5] also offers a mobile application, but is chained via internet to the power of a desktop computer. In this paper we develop a filter design toolbox for mobile devices. The mathematics developed had to be compatible with the limited memory and processing power of these devices. It provides filter design, real time playback, and frequency/impulse response plotting. Versions of this toolbox were created for the iPhone and iPad.

II. SOFTWARE DESIGN

This section describes the IIR filter design with Biquad form using Bilinear transformation. There are 3 steps, i.e. Lowpass prototype design, frequency transformation, and bilinear transformation. The derivation for mobile implementation is discussed. This new method fits for the small memory constrain on the mobile devices and yield the same results as other PCs softwares, e.g. Matlab’s Fdatool.

A. Analog Prototype Lowpass Filter

The goal of this step is to design the analog lowpass filter with cutoff frequency equals to one. The final equation, using any design, will be in this form, where \( N_p \) is the number of biquad (second order) states.

\[
H(s) = G_s \prod_{k=0}^{N_p-1} \frac{b_{2k} s^2 + b_{2k-1} s + b_{2k-2}}{a_{2k} s^2 + a_{2k-1} s + a_{2k-2}}
\]  

(1)

Three methods were designed in the tool, i.e. Butterworth, Chebyshev I and II, to estimate \( G_s \), \( B_s \), and \( A_s \) parameters. Given the filter order \( N \), the simplified transfer function \( H(s) \) for Butterworth and Chebyshev I filter can be expressed as following:

\[
H(s) = \frac{G_s}{(s + a)} \prod_{i=0}^{N-1} \frac{1}{s^2 - 2a \cos \left( \frac{N + 1}{2N} \pi + \frac{k \pi}{N} \right) s + |p_i|^2}
\]  

(2)

For the Butterworth filter, since pole locations are located on the unit circle, \( a, G_s, |p_i|^2 \) become one in the equation (2). For the Chebyshev I, since the poles are not placed on the unit circle but instead form an oval shape center at the origin, \( a, G_s, |p_i|^2 \) parameters depend on the ripple magnitude of the passband. For both cases, if the order \( N \) is an odd
number, r becomes one and when the order N is even, r becomes zero. Given the passband ripple magnitude \( R_p \) in linear scale, the rest of the parameters can be estimated by

\[
|p_i|^2 = a^2 \cos \left( \frac{N+1}{2N} \pi + \frac{k\pi}{N} \right) + b^2 \sin \left( \frac{N+1}{2N} \pi + \frac{k\pi}{N} \right),
\]

where

\[
a = \frac{1}{2} \left( \sqrt{\alpha} - \sqrt{1/\alpha} \right), \quad b = \frac{1}{2} \left( \sqrt{\alpha} + \sqrt{1/\alpha} \right).
\]

Finally, the gain \( G_s \) of Chebyshev I filter is computed.

\[
G_s = a^r \left( \frac{R_p}{s} \right)^{r-1} \prod_{k=0}^{N-1} |p_k|^2
\]

For Chebyshev II, the ripple occurs in the stopband frequency. Given the stopband ripple magnitude \( R_s \) in linear scale, the simplified \( H(s) \) of the Chebyshev II can be expressed as following:

\[
H(s) = \frac{G_s \prod_{k=1}^{N} s^2 + \frac{1}{\cos^2 \left( \frac{(2k-1)\pi}{2N} \right)}}{\left( s - \frac{1}{e} \right) \prod_{k=N+1}^{2N} s^2 - \frac{2c_k}{c_k + d_k} s + \frac{1}{c_k + d_k} \Gamma}.
\]

\[
\Gamma = \left( \sqrt{R_s + \sqrt{R_s - 1}} \right)^{1/N}
\]

\[
c_k = \frac{\Gamma^2 - 1}{2\Gamma} \sin \left( \frac{(2k-1)\pi}{2N} \right)
\]

\[
d_k = \frac{\Gamma^2 + 1}{2\Gamma} \cos \left( \frac{(2k-1)\pi}{2N} \right)
\]

where \( e \) is equal to \( c_k \) with \( k = \left[ \frac{3N+2}{2} \right] \). Finally, the gain \( G_s \) of Chebyshev II filter is computed by the ratio of the product of all \( a_{2k} \) and the product of all \( b_{2k} \),

\[
G_s = \left( \prod_{k} a_{2k} \right) \left( \prod_{k} b_{2k} \right)^{-1}
\]

Parameters are 0.5 dB (or 1.122 in linear scale) and 30 dB (or 1000 in linear scale) for Chebyshev I and II, respectively.

![Fig. 1. Screenshots of analog lowpass prototype outputs from iOS implementation using (a) Butterworth, (b) Chebyshev I, and (c) Chebyshev II methods. All the results are identical with Matlab.](image)

**B. Frequency Transformation**

The goal of this step is to transform the lowpass filter with cut-off at one to one of four filter types, i.e. lowpass, highpass, bandpass, bandstop, at the desired cut-off frequency. The final form after transformation to lowpass and highpass filters will be in this form.

\[
H(s) \rightarrow H_c(s) = G \prod_{k=0}^{N-1} b_{2k}d_{2k}^2 + b_{2k}'u + b_{2k}''u^2 + a_{2k}d_{2k}^2 + a_{2k}'u + a_{2k}''u^2
\]

Note that the gain and number of biquads does not change during this step. The final outputs are still in the biquad form. The implementation in this step is to estimate \( B' \) and \( A' \) parameters. After derivation, the solution becomes a simple substitution of the previous step. For the lowpass filter with cut-off frequency \( \Omega \) Hz, the mapping equation can be expressed as

\[
b_{k0} = b_{k0}' \quad b_{k1} = b_{k1}' \Omega \quad b_{k2} = b_{k2}' \Omega^2
\]

\[
a_{k0} = a_{k0}' \quad a_{k1} = a_{k1}' \Omega \quad a_{k2} = a_{k2}' \Omega^2
\]

On the other hand, for highpass frequency with cut-off frequency \( \Omega \) Hz, the mapping equation can be expressed as

\[
b_{k0} = b_{k2}' \quad b_{k1} = b_{k2}' \Omega \quad b_{k2} = b_{k2}' \Omega^2
\]

\[
a_{k0} = a_{k2}' \quad a_{k1} = a_{k2}' \Omega \quad a_{k2} = a_{k2}' \Omega^2
\]
transformation in the next step. For the frequency transformation of the bandpass filter with cut-off frequency $\Omega$ Hz. and bandwidth $\Delta$ Hz., can be expressed as following:

$$H(s) = \rightarrow H(z) = G_i \prod_{k=0}^{n} \frac{b'_{k0} s^2 + b'_{k1} s + b'_{k2}}{a'_{k0} s^2 + a'_{k1} s + a'_{k2}}$$

(9).

The mapping function for the bandpass filter can be expressed as following.

$$b'_{k0} = b_{k0}$$
$$a'_{k0} = a_{k0}$$
$$b'_{k1} = b_{k1}$$
$$a'_{k1} = a_{k1}$$
$$b'_{k2} = 2b_{k0} \Omega^2 + b_{k2} \Delta^2$$
$$a'_{k2} = 2a_{k0} \Omega^2 + a_{k2} \Delta^2$$
$$b'_{k3} = b_{k1} \Omega^2$$
$$a'_{k3} = a_{k1} \Omega^2$$
$$b'_{k4} = b_{k2} \Omega^4$$
$$a'_{k4} = a_{k2} \Omega^4$$

(10).

For the frequency transformation of the bandstop filter, the equation is simply the interchange between $b_{k0}$, $a_{k0}$ and $b_{k2}$, $a_{k2}$ of the bandpass filter.

$$b'_{k0} = b_{k2}$$
$$a'_{k0} = a_{k2}$$
$$b'_{k1} = b_{k1}$$
$$a'_{k1} = a_{k1}$$
$$b'_{k2} = 2b_{k2} \Omega^2 + b_{k0} \Delta^2$$
$$a'_{k2} = 2a_{k2} \Omega^2 + a_{k0} \Delta^2$$
$$b'_{k3} = b_{k2} \Omega^2$$
$$a'_{k3} = a_{k2} \Omega^2$$
$$b'_{k4} = b_{k2} \Omega^4$$
$$a'_{k4} = a_{k2} \Omega^4$$

(11).

The cut-off frequency $\Omega$ and bandwidth $\Delta$ are the prewrapping frequency. Given the sampling frequency $F_s$ and desired cut-off frequency $F_c$ in digital domain, the prewrapping frequency is estimated by $\Omega = 2F_s \tan(\pi F_c / F_s)$. When using them in bandpass and bandstop, The cut-off frequency $\Omega$ is the geometric mean of the high and low prewrapping frequencies, and the bandwidth $\Delta$ is the difference between two frequencies.

C. Bilinear Transformation

The goal of this final step is to transform from analog domain to digital domain ($z$ domain). For the lowpass and highpass biquad, the final form is similar to the previous step, except the mapping function is different. Again the gain and number of biquad does not change during this step.

$$H_s(u) = \rightarrow H(z) = G \prod_{k=0}^{n} \frac{b_{k0} + b_{k1}z^{-1} + b_{k2}z^{-2}}{a_{k0} + a_{k1}z^{-1} + a_{k2}z^{-2}}$$

(12).

The mapping function for the biquad form using bilinear transformation can be expressed as following:

$$\hat{b}_{k0} = 4b_{k0}F_s^2 + 2b_{k1}F_s + b_{k2}$$
$$\hat{a}_{k0} = 4a_{k0}F_s^2 + 2a_{k1}F_s + a_{k2}$$
$$\hat{b}_{k1} = -8b_{k1}F_s^2 + 2b_{k2}$$
$$\hat{a}_{k1} = -8a_{k1}F_s^2 + 2a_{k2}$$
$$\hat{b}_{k2} = 4b_{k2}F_s^2 - 2b_{k1}F_s + b_{k2}$$
$$\hat{a}_{k2} = 4a_{k2}F_s^2 - 2a_{k1}F_s + a_{k2}$$

(13).

For the bandpass and bandstop, since they are fourth order, the transformation can be expressed as following:

$$H(z) = G \prod_{k=0}^{n} \frac{b_{k0} + b_{k1}z^{-1} + b_{k2}z^{-2} + b_{k3}z^{-3} + b_{k4}z^{-4}}{a_{k0} + a_{k1}z^{-1} + a_{k2}z^{-2} + a_{k3}z^{-3} + a_{k4}z^{-4}}$$

(14).

The mapping function for the fourth form using bilinear transformation can be expressed as following.

$$\hat{b}_{k0} = 16b_{k0}F_s^4 + 8b_{k1}F_s^3 + 4b_{k2}F_s^2 + 2b_{k3}F_s + b_{k4}$$
$$\hat{b}_{k1} = -64b_{k1}F_s^4 - 16b_{k2}F_s^3 + 4b_{k3}F_s^2 + 4b_{k4}$$
$$\hat{b}_{k2} = 96b_{k2}F_s^4 - 8b_{k3}F_s^3 + 4b_{k4}$$
$$\hat{b}_{k3} = -64b_{k3}F_s^4 + 16b_{k4}F_s^3 - 8b_{k2}F_s^2 + 4b_{k4}$$
$$\hat{b}_{k4} = 16b_{k4}F_s^4 - 3b_{k2}F_s^3 + 4b_{k4}$$

(15).

For the $A$ parameters, the mapping function is the same fashion of the above equation by interchanging $b$ to $a$ terms. Once the bilinear transformation is finished, the decomposition back to biquad is applied. The task is done by finding 4 roots of $B$ and $A$ parameters and pairing the same complex conjugate terms. This will make the number of biquads double. The final normalization is done by making $a_{k0}$ and $b_{k0}$ equal to one. Fig. 2 shows the results of digital IIR filters running on an iPhone.
Fig. 3. Screenshots of mobile digital filter design toolbox on iOS devices: (a) main menu on an iPhone app where the user can choose each method, (b) the email system where the user can send the design coefficients in floating and fixed point formats, and (c) graphing on an iPad app.

III. RESULTS

This mobile digital filtering toolbox was implemented on iOS devices, both iPhone and iPad. The iOS SDK version 4.3 was used in this project. The OS is comprised of 4 layers including the core OS, Core Services, Media, and Cocoa Touch layers. Most of the programs in this paper were implemented using Cocoa Touch and Objective-C code. Cocoa Touch is the set of Objective-C frameworks that provide the building blocks for iOS applications. Contained are all the user interface widgets, events and event loop management, APIs to respond to touch, gestures, movement, access to the camera, file system, and other device features. Fig. 3 illustrates the screenshots of mobile apps running on both devices. Due to the screen size, the additional features such as real-time audio playback, frequency response plotting are not implemented in every device. On the iPad app (DSP Assistant app [6] on the app store®), the program can plot the impulse and frequency responses of the designing filter. The results can be shown in both floating and fixed point formats (16 bits) and sent out via email. The iPhone app (DSP digital filter app [7] on the app store®) functions identically to the iPad app, except it does not plot. The iPhone program can also playback the audio passed thru the filter in real-time. Users can turn on/off the filter function in real-time. This mobile tool can also estimate FIR filter coefficients, which we did not discuss the design in this paper. There are 6 different types, e.g. same 4 types as IIR and the additional Hibert, Differentiator, of FIR filters and 8 window types, e.g. Kaiser, Gaussian, Hanns, that can be selected. The plotting, email function, both fixed and floating formats, are similar to IIR tools. The implementation on Android mobiles is ongoing.

IV. CONCLUSION

A filter design toolbox was designed and implemented for mobile devices. Different versions were created for the iPhone and iPad devices because of differences in screen size. The iPhone version allows a user to design a filter and filter real time audio playback. The iPad version is identical, with addition of plotting functions for both impulse and frequency responses. There are multiple filter types and windows that can be implemented in the system. The toolbox supports the design of both FIR and IIR filter types. Once a filter is designed, the application allows users to email the filter coefficients in a format that can be implemented in Matlab, along with the order, type of filter and other pertinent information.

REFERENCES

function [Bc,Ac,G] = ButterAnalog(N);
%% Usage: [Bc,Ac,G] = ButterAnalog(N);
%% Design the normalized analog butterworth filter prototype with OMEGA = 1
%% input is N (order): scalar
%% output Biquad coefficients with B, A, vectors and G

%%%%%%%%%%%%%%%%% STEP 1 %%%%%%%%%%%%%%%%%%%%%%%
%%%% Create a output martix %%%%%%%%%%%%%%%%%%%
Ac = zeros(ceil(N/2),3);   %% prepare the matrix
Bc = zeros(ceil(N/2),3);   %% prepare the matrix
G = 1;

%%%%%%%%%%%%%%%%% STEP 2 %%%%%%%%%%%%%%%%%%%%%%%
%%%% make the incomplete biquad if N = odd %%%%%%

%%%%%%%%%%%%%%%%% STEP 3 %%%%%%%%%%%%%%%%%%%%%%%
%%%% Compute the complete biquad %%%%%%%%%%%%%%%
for k = 0:floor(N/2)-1
    Bc(k+1,3) = 1;   %% the numerator is always 1/Ac
    Ac(k+1,1) = 1;   %% s^2
    Ac(k+1,3) = 1;   %% 1
    Ac(k+1,2) = -2*cos(((N+1)*pi/(2*N))+k*pi/N);
end
if (rem(N,2) == 1)  %% coefficients for s, even case
    k = floor(N/2);
    Bc(k+1,3) = 1;   %% the numerator is always 1/Ac
    Ac(k+1,1) = 0;   %% s^2
    Ac(k+1,3) = 1;   %% 1
    Ac(k+1,2) = 1;
end
function [Bc,Ac,G] = Cheby1Analog(N,Rp);

%% usage: [Bc,Ac,G] = Cheby1Analog(N,Rp);
%% design the normalized analog cheby1 filter prototype
%% input is N (order): scalar
%%          Rp: ripple value, default at 0.5
%% output is SOS2 (B,A) and Gain for biquad: matrix

%% make a default for Rp
if nargin == 1
    Rp = 0.5;
end

Rp = 10^(Rp/10);               % linear

Ac = zeros(ceil(N/2),3);       % prepare the matrix
Bc = zeros(ceil(N/2),3);       % prepare the matrix
G = 1;

% compute alpha
alpha = (1+sqrt(Rp))/sqrt(Rp-1);
%% compute a and b
a = (1/2)*((alpha^(1/N))-(1/alpha)^(1/N));
b = (1/2)*((alpha^(1/N))+((1/alpha)^(1/N)));

%% start to compute biquad
for k = 0:floor(N/2)-1
    pr = a*cos((N+1)*pi/(2*N) + k*pi/N);
    pii = b*sin((N+1)*pi/(2*N) + k*pi/N);
    Bc(k+1,3) = 1;                % the numerator is always 1.
    Ac(k+1,1) = 1;                % pair with conjugate pole, S^2
    Ac(k+1,2) = -2*pr;            % 2*re{p}S
    Ac(k+1,3) = (pr.^2)+(pii.^2);  % abs(p)^2
    G = G*Ac(k+1,3);
end
if (rem(N,2) == 1)
    K = floor(N/2);
    Bc(K+1,3) = 1;                % the numerator is always 1.
    Ac(K+1,1) = 0;                % pair with conjugate pole, S^2
    Ac(K+1,2) = 1;                % 2*re{p}S
    Ac(K+1,3) = a;                % abs(p)^2
end

G = (G*(a)^(rem(N,2)))/(Rp)^((1-rem(N,2))/2);
function [Bc,Ac,G] = Cheby2Analog(N,Rs)
%% usage: [Bc,Ac,G] = Cheby2Analog(N,Rs);
%% design the normalized analog cheby2 filter prototype
%% input is N (order): scalar
%% Rs: ripple value, default at 30
%% output is SOS2 (B,A) and Gain for biquad: matrix

%% make a default for Rp
if nargin == 1
    Rs = 30;
end

Ws = 1;
Rs = 10^(Rs/10);
gm = (sqrt(Rs)+sqrt(Rs-1));
Gr = gm^(1/N);
G = 1;

ctr = 1;
for k = N+1:floor(3*N/2);
a = sin(((2*k)-1)*pi/(2*N));
b = cos(((2*k)-1)*pi/(2*N));
a = a*((Gr^2)-1)/(2*Gr);
b = b*((Gr^2)+1)/(2*Gr);
Ac(ctr,:) = [1 1-2*a/(a*a+b*b) 1/(a*a+b*b)];
G = G*Ac(ctr,3);
ctr = ctr+1;
end
if (rem(N,2) == 1)
    k = floor((3*N+2)/2);
    a = sin(((2*k)-1)*pi/(2*N));
    a = a*((Gr^2)-1)/(2*Gr);
    Ac(ctr,:) = [0 1 1/a];
    G = G*Ac(ctr,3);
end

ctr = 1;
for k = 1:floor(N/2)
    b = cos(((2*k)-1)*pi/(2*N));
    Bc(ctr,:) = [1 0 1/(b*b)];
    G = G/Bc(ctr,3);
    ctr = ctr+1;
end

%% test code with built-in matlab code
%% [BB,AA] = cheby2(N,30,1,'s');
%% [SOS,G] = tf2sos(BB,AA)
function [Bt,At] = LowpassToLowpass(B,A,Omega);
%% Usage: [Bt,At] = LowpassToLowpass(B,A,Omega);
%% mapping a prototype LP to a specific LP with correct cutoff frequency.
%% inputs are B, A vectors (1 by 3) for each biquad state, Omega is freq cutoff.
%% outputs are Bt, At vectors (1 by 3) for each biquad state.

b0 = B(1);   %% s^2
b1 = B(2);   %% s
b2 = B(3);   %% 1
a0 = A(1);   %% s^2
a1 = A(2);   %% s
a2 = A(3);   %% 1

if (a0 == 0)   % incomplete biquad
    Bt(1) = 0;
    Bt(2) = b1;
    Bt(3) = b2*Omega;
    At(1) = 0;
    At(2) = a1;
    At(3) = a2*Omega;
else            % complete biquad
    Bt(1) = b0;
    Bt(2) = b1*Omega;
    Bt(3) = b2*Omega*Omega;
    At(1) = a0;
    At(2) = a1*Omega;
    At(3) = a2*Omega*Omega;
end
function [Bt,At] = LowpassToHighpass(B,A,Omega);
%% Usage: [Bt,At] = LowpassToHighpass(B,A,Omega);
%% mapping a prototype LP to a specific HP with correct cutoff frequency.
%% inputs are B, A vectors (1 by 3) for each biquad state, Omega is freq cutoff.
%% outputs are Bt, At vectors (1 by 3) for each biquad state.

b0 = B(1);    %% s^2
b1 = B(2);    %% s
b2 = B(3);    %% 1
a0 = A(1);    %% s^2
a1 = A(2);    %% s
a2 = A(3);    %% 1

if (a0 == 0)    %% incomplete biquad
    Bt(1) = 0;
    Bt(2) = b2;
    Bt(3) = b1*Omega;
    At(1) = 0;
    At(2) = a2;
    At(3) = a1*Omega;
else            %% complete biquad
    Bt(1) = b2;
    Bt(2) = b1*Omega;
    Bt(3) = b0*Omega*Omega;
    At(1) = a2;
    At(2) = a1*Omega;
    At(3) = a0*Omega*Omega;
end
function \([Bt,At] = \text{LowpassToBandpass}(B,A,\Omega,\text{BW});\)

%% Usage: \([Bt,At] = \text{LowpassToBandpass}(B,A,\Omega,\text{BW});\)
%% mapping a prototype LP to a specific BP with correct cutoff frequency.
%% inputs are B, A vectors (1 by 3) for each biquad state, \(\Omega\) is freq cutoff and \(\text{BW}\)
%% is bandwidth.
%% outputs are \(Bt, At\) vectors (1 by 3) for each biquad state.

\[\begin{align*}
  b0 &= B(1); \quad \% \ s^2 \\
  b1 &= B(2); \quad \% \ s \\
  b2 &= B(3); \quad \% \ 1 \\
  a0 &= A(1); \quad \% \ s^2 \\
  a1 &= A(2); \quad \% \ s \\
  a2 &= A(3); \quad \% \ 1 \\
\end{align*}\]

if \(a0 == 0\) \hspace{1cm} \% incomplete biquad
  \[\begin{align*}
    Bt(1) &= 0; \\
    Bt(2) &= 0; \\
    Bt(3) &= b1; \\
    Bt(4) &= b2*\text{BW}; \\
    Bt(5) &= b1*\Omega*\Omega; \\
    At(1) &= 0; \\
    At(2) &= 0; \\
    At(3) &= a1; \\
    At(4) &= a2*\text{BW}; \\
    At(5) &= a1*\Omega*\Omega; \\
  \end{align*}\]
else \hspace{1cm} \% complete biquad
  \[\begin{align*}
    Bt(1) &= b0; \\
    Bt(2) &= b1*\text{BW}; \\
    Bt(3) &= (2*b0*\Omega*\Omega)+(b2*\text{BW}^2); \\
    Bt(4) &= b1*\text{BW}^2*\Omega*\Omega; \\
    Bt(5) &= b0*\Omega^4; \\
    At(1) &= a0; \\
    At(2) &= a1*\text{BW}; \\
    At(3) &= (2*a0*\Omega*\Omega)+(a2*\text{BW}^2); \\
    At(4) &= a1*\text{BW}^2*\Omega*\Omega; \\
    At(5) &= a0*\Omega^4; \\
  \end{align*}\]
end
function [Bt,At] = LowpassToBandstop(B,A,Omega,BW);
%% Usage: [Bt,At] = LowpassToBandstop(B,A,Omega,BW);
%% mapping a prototype LP to a specific BP with correct cutoff frequency.
%% inputs are B, A vectors (1 by 3) for each biquad state, Omega is freq cutoff and BW
%% is bandwidth.
%% outputs are Bt, At vectors (1 by 3) for each biquad state.

b0 = B(1);   %% s^2
b1 = B(2);   %% s
b2 = B(3);   %% 1
a0 = A(1);   %% s^2
a1 = A(2);   %% s
a2 = A(3);   %% 1

if (a0 == 0)          %% incomplete biquad
    Bt(1) = 0;
    Bt(2) = 0;
    Bt(3) = b2;
    Bt(4) = b1*BW;
    Bt(5) = b2*(Omega^2);
    At(1) = 0;
    At(2) = 0;
    At(3) = a2;
    At(4) = a1*BW;
    At(5) = a2*(Omega^2);
else                  %% complete biquad
    Bt(1) = b2;
    Bt(2) = b1*BW;
    Bt(3) = (2*b2*Omega*Omega)+(b0*BW*BW);
    Bt(4) = b1*BW*Omega*Omega;
    Bt(5) = b2*(Omega^4);
    At(1) = a2;
    At(2) = a1*BW;
    At(3) = (2*a2*Omega*Omega)+(a0*BW*BW);
    At(4) = a1*BW*Omega*Omega;
    At(5) = a2*(Omega^4);
end
function [bb,aa] = BilinearZTransform2(B,A,Fs);

%% usage: function [bb,aa] = BilinearZTransform2(B,A,Fs);
%% Bilinear transform for order 2
%% input is B,A : vector (1 by 3)
%% Fs: a sampling rate
%% output is bb and aa in z domain

b0 = B(1);    %% s^2
b1 = B(2);    %% s
b2 = B(3);    %% 1
a0 = A(1);    %% s^2
a1 = A(2);    %% s
a2 = A(3);    %% 1
alp = 2*Fs;
alp_sq = alp^2;

if (a0 == 0)   %% first order
    bb(1) = (b1*alp)+b2;
    bb(2) = (-b1*alp)+b2;
    bb(3) = 0;
    aa(1) = (a1*alp)+a2;
    aa(2) = (-a1*alp)+a2;
    aa(3) = 0;
else            %% second order
    bb(1) = (4*b0*Fs*Fs)+(2*b1*Fs)+b2;
    bb(2) = (-8*b0*Fs*Fs)+(2*b2);
    bb(3) = (4*b0*Fs*Fs)-(2*b1*Fs)+b2;
    aa(1) = (4*a0*Fs*Fs)+(2*a1*Fs)+a2;
    aa(2) = (-8*a0*Fs*Fs)+(2*a2);
    aa(3) = (4*a0*Fs*Fs)-(2*a1*Fs)+a2;
end

bb = bb/aa(1);   %% normalize aa0 to be 1
aa = aa/aa(1);
function [bb,aa] = BilinearZTransform4(B,A,Fs);
流行: function [bb,aa] = BilinearZTransform4(B,A,Fs);
%% Bilinear transform for order 2
%% input is B,A : vector (1 by 5)
%% Fs: a sampling rate
%% output is bb and aa in z domain

b0 = B(1);  %% s^4
b1 = B(2);  %% s^3
b2 = B(3);  %% s^2
b3 = B(4);  %% s
b4 = B(5);  %% 1
a0 = A(1);  %% s^4
a1 = A(2);  %% s^3
a2 = A(3);  %% s^2
a3 = A(4);  %% s
a4 = A(5);  %% 1

alp   = 2*Fs;
alp_2 = alp^2;
alp_3 = alp^3;
alp_4 = alp^4;

if  ((a0 == 0) && (a1 == 0))  % second order
    bb(1) = (b2*alp_2)+(b3*alp)+b4;
    bb(2) = (-2*b2*alp_2)+(2*b4);
    bb(3) = (b2*alp_2)-(b3*alp)+b4;
    bb(4) = 0;
    bb(5) = 0;
    aa(1) = (a2*alp_2)+(a3*alp)+a4;
    aa(2) = (-2*a2*alp_2)+(2*a4);
    aa(3) = (a2*alp_2)-(a3*alp)+a4;
    aa(4) = 0;
    aa(5) = 0;
else
    % 4th order
    bb(1) = (16*b0*(Fs^4)) +(8*b1*(Fs^3)) +(4*b2*(Fs^2)) +(2*b3*Fs) + b4;
    bb(2) = (-64*b0*(Fs^4)) +(-16*b1*(Fs^3)) +0 +(-8*b2*(Fs^2)) +0 + (6*b4);
    bb(3) = (96*b0*(Fs^4)) +0 +(8*b2*(Fs^2)) +0 + (6*b4);
    bb(4) = (-64*b0*(Fs^4)) +(16*b1*(Fs^3)) +0 +(4*b3*Fs) + (4*b4);
    bb(5) = (16*b0*(Fs^4)) +(8*b1*(Fs^3)) +(4*b2*(Fs^2)) +(-2*b3*Fs) + b4;
    aa(1) = (a0*alp_4) +(a1*alp_3) +(a2*alp_2) +(a3*alp) + a4;
    aa(2) = (-4*a0*alp_4)+(2*a1*alp_3)+0 +(-2*a2*alp_2)+0 +(6*a4);
    aa(3) = (6*a0*alp_4) +0 +(-2*a2*alp_2)+0 + (6*a4);
    aa(4) = (-4*a0*alp_4)+(2*a1*alp_3)+0 +(-2*a3*alp)+ (4*a4);
    aa(5) = (a0*alp_4) +(a1*alp_3) +(a2*alp_2) +(a3*alp) + a4;
end

bb = bb/aa(1);  % normalize aa0 to be 1
aa = aa/aa(1);