Digital Equalization for Audio with Application to Loudspeakers and Rooms

by

Marc E. Bonneville, B.A.Sc.

A thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute for Electrical Engineering

Department of Electrical Engineering
Faculty of Engineering
University of Ottawa

January, 1992.

Copyright © 1992 Marc E. Bonneville
Patent Application Pending

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-15700-8

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

The application of a non-minimum-phase digital filterbank as a graphic equalizer is investigated. The filterbank effected a timbral modification of some musical signals even though the magnitude response was unity. For this reason, the filterbank was rejected as a candidate for a high-fidelity graphic equalizer.

A new type of equalizing filter is introduced in this work. This filter has the property that its log-magnitude frequency response is approximately linear in the gain term. This filter is used as a building block for an analog and digital parametric equalizer whose frequency response can be adjusted automatically for the use of loudspeaker equalization.

Acknowledgements

This work was made possible in part through financial support from the Natural Sciences and Engineering Research Council and the Canadian Audio Research Consortium.

I wish to thank my supervisor Dr. Bill McGee and the members of the Canadian Audio Research Consortium especially Dr. Claude Fortier, Ian Paisley and Dr. Eric Verreault. I also wish to thank the people in the department of Acoustics and Signal Processing at the National Research Council: Sean Olive, Dr. Floyd Toole and especially Dr. Peter Schuck.

Finally, I wish to thank my wife Brigitte and all those who have helped me in any way during the past year.

Table of Contents

List of Figures v

List of Tables vii

List of Terms and Abbreviations viii

Chapter 1

1. Introduction 1
- 1.1. Problem Statement 1
- 1.2. Background 1
  - 1.2.1. Graphic Equalizers 2
    - 1.2.1.1. Analog Graphic Equalizers 3
      - 1.2.1.1.1. Topology 3
      - 1.2.1.1.2. Boost and Cut Symmetry 7
      - 1.2.1.1.2. Bandpass Filter Order and Q 12
      - 1.2.1.1.3 Phase Response 14
    - 1.2.1.2. Digital Graphic Equalizers 16
  - 1.2.2. Parametric Equalizers 17
    - 1.2.2.1. Analog Parametric Equalizers 18
    - 1.2.2.2. Digital Parametric Equalizers 19
  - 1.2.3. Digital Filter Design Techniques 20
  - 1.2.4. Adaptive Filters 20
- 1.3. Scope 21

Chapter 2

2. Allpass Filterbank 22
- 2.1. Analytic Frequency Response 27
  - 2.1.2. Phase Response 32

2.2. Floating Point Simulation 39
2.2.1. Impulse Response 39
2.2.2. Subjective Listening Tests 39

Chapter 3

3. Parametric Equalizer 46
3.1. Bode Equalizer 47
3.1.1. Background 47
3.1.2. Biquadratic Equalizing Filters 50
3.1.2.1. Error Analysis 53
3.1.2.2. Phase Response 55
3.1.2.3. Changing Q Value 57
3.1.2.4. Alternate-Error-Measure 58
3.2. Modified-Bode Equalizer 61
3.2.1. Biquad Approximation 62
3.2.1.1. Error Analysis 64
3.2.1.2. Alternate-Error-Measure 67
3.2.1.3. Optimum Scaling Factor 69
3.2.2. First-Order Shelving Approximation 72
3.2.2.1. Error Analysis 75
3.2.3. Automation Method 1 - Biquads 76
3.2.4. Automation Method 2 - Biquads 79
3.2.4.1. Gain Term 79
3.2.4.2. Center Frequencies 80
3.2.4.3. Q 81
3.2.4.4. Frequency Response Weighting 81
3.2.4.5. General Description 82
3.2.5. Automation Method 2 - Shelving 85
3.2.5.1. Gain Term 85
3.2.5.2. Center Frequencies 85
3.2.6. Frequency Response Measurement 85
3.2.6.1. Frequency Response Smoothing 85
3.2.6.2. Target Frequency Response 86

3.2.6.3. Normalization....88
3.2.7. Cascaded Versus Paralleled Sections....89
3.2.8. Global Automation Algorithm - Biquads....89
3.2.9. Global Automation Algorithm - Shelving....94
3.2.10. Biquad Simulations....94
3.2.10.1. Subwoofer....94
3.2.10.2. Satellite Speaker....99
3.2.11. Shelving Simulation....102
3.2.12. Subjective Tests....102
3.2.13. Digital Case....104
3.2.13.1. Frequency Warping Pre-Processing....105
3.2.13.2. Digital Biquad Parameters....105
3.2.13.3. Digital Shelving Parameters....106
3.2.13.4. Biquad Simulation....107
3.2.13.5. Shelving Simulation....110

Chapter 4

4. Conclusions and Recommendations....111
4.1. Conclusions....111
4.2. Recommendations for Further Work....114

References....116

Appendix

Software Listings....121
Appendix A - Filterbank Frequency Response....121
Appendix B - Filterbank Floating Point Simulation....136
Appendix C - Empirical Optimum Scaling Factor....142
Appendix D - Automation Algorithms....150
Biquadratic Filter....150
Shelving Filter....158

Warping......164
Dewarping......166
Target Frequency Response......168
Functions......171

List of Figures

Figure 1.1 Analog Graphic Equalizer Block Diagram....................................3
Figure 1.2 Unsymmetrical Boost and Cut Equalizer Response........................6
Figure 1.3 Unsymmetrical Boost and Cut Pole-Zero Locations.......................7
Figure 1.4 Symmetrical Boost and Cut Graphic Equalizer Schematic...........8
Figure 1.5 Symmetrical Boost and Cut Pole-Zero Locations..........................10
Figure 1.6 Symmetrical Boost and Cut Equalizer Response...........................11
Figure 1.7 Analog Parametric Equalizer Schematic.....................................19
Figure 2.1 Filterbank Magnitude Response of one Stage.............................23
Figure 2.2 Filterbank Block Diagram...26
Figure 2.3 Allpass Filterbank Frequency Response....................................28
Figure 2.4 Unwrapped Phase Response...29
Figure 2.5 One Weight at -10 dB...29
Figure 2.6 Frequency Response with High-Frequency Weights at -12 dB.31
Figure 2.7 Allpass Filterbank Phase Delay...34
Figure 2.8 Allpass Filterbank Group Delay...34
Figure 2.9 Real-Output-Weights Filterbank With High-Frequency Weights at -12 dB...37
Figure 2.10 Allpass Filterbank Impulse Response..39
Figure 2.11 Allpass Filterbank Input and Output Waveforms for Track 1....41
Figure 2.12a Allpass Filterbank Input Waveform for Track 2.........................42
Figure 2.12b Allpass Filterbank Output Waveform for Track 2.......................42
Figure 3.1 Bode Equalizer Error...55
Figure 3.2 Bode Equalizer Typical Frequency Response Curves..................56
Figure 3.3 Bode Equalizer Pole-Zero Positions...57
Figure 3.4 Alternate-Error-Measure: Bode...60
Figure 3.5 Alternate-Error-Measure: Bode and Regular Biquad.................61
Figure 3.6 Modified-Bode Equalizer Typical Frequency Response Curves.66
Figure 3.7 Modified-Bode Biquad Equalizer Pole-Zero Locations.................67
Figure 3.8 Modified-Bode Equalizer Error...68
Figure 3.9 Alternate-Error-Measure: Bode, Biquad and Modified-Bode......69

Figure 3.10 Optimum Error Parameter Values 72
Figure 3.11 Modified-Bode First-Order Equalizer Pole-Zero Locations 73
Figure 3.12 First-Order Shelving Frequency Response 74
Figure 3.13 First-Order Shelving Error Curve 76
Figure 3.14a Best Biquad Stage Algorithm 83
Figure 3.14b Best Biquad Stage Algorithm 84
Figure 3.15 Input Data Measurement Algorithm 87
Figure 3.16 Target Frequency Response Algorithm 88
Figure 3.17a Global Algorithm 91
Figure 3.17b Global Algorithm 92
Figure 3.17c Global Algorithm 93
Figure 3.18 Subwoofer-Room Magnitude Response (FFT) 95
Figure 3.19 Subwoofer-Room Smoothed Magnitude Response 95
Figure 3.20 Subwoofer-Room Target and Equalized Magnitude Response 96
Figure 3.21 Exact Subwoofer Equalizer Response of Each Stage 97
Figure 3.22 Exact Total Subwoofer Equalizer Response 97
Figure 3.23 Approximate Total Subwoofer Equalizer Response 98
Figure 3.24 Total Subwoofer Equalizer Response Approximation Error 98
Figure 3.25 Satellite Speaker-Room Magnitude Response 99
Figure 3.26 Satellite Equalized Speaker-Room Response and Target 100
Figure 3.27 Exact Satellite Equalizer Response of Each Stage 101
Figure 3.28 Exact Total Satellite Equalizer Response 101
Figure 3.29 Equalized Subwoofer Response Using One Shelving Stage 103
Figure 3.30 Shelving Subwoofer Equalizer Response 103
Figure 3.31 Pre-Warped Satellite and Target Responses 108
Figure 3.32 Digitally Equalized Satellite Speaker Response 108
Figure 3.33 Exact Digital Equalizer Response of Each Stage 109
Figure 3.34 Exact Total Digital Satellite Equalizer Response 110

List of Tables

Table 2.1 All-Pass Filterbank Coefficients..30
Table 2.2 Minimum Phase Filterbank Coefficients..38
Table 3.1 Subwoofer Equalizer Filter Parameters..96
Table 3.2 Satellite Equalizer Filter Parameters..100
Table 3.3 Digital Satellite Equalizer Filter Parameters..................................109

List of Terms and Abbreviations

Alternate-Error-Measure : $error_{aem}(\omega)$. The difference between one magnitude frequency response and a scaled version of another.

Biquad : biquadratic function. Characterized by a second-order numerator and a second-order denominator.

Bode Biquad : $$H_{Bd}(s) = \frac{s^2 + s\,[1 + yG]\,\frac{\omega_o}{Q} + \omega_o^2}{s^2 + s\,[1 - yG]\,\frac{\omega_o}{Q} + \omega_o^2}$$

dB : decibel = 20 x log-magnitude where the base 10 logarithm is used.

dBSPL : sound pressure level in dB with respect to 20μPa.

DSP : digital signal processing.

FIR : finite impulse response.

FFT : fast Fourier transform.

Group Delay : derivative of the phase with respect to frequency times -1.

$$\tau_g = -\frac{d\phi}{d\omega}$$

IIR : infinite impulse response.

LMS : least-mean-square.

Log-Magnitude : the logarithm of the magnitude frequency response. When the natural logarithm is used, the units are Nepers.

Minimum-Phase System : a system whose phase and log-magnitude response form a Hilbert transform pair.

Modified-Bode Biquad : $H_{mBd}(s) = \dfrac{s^2 + s\dfrac{K\omega_o}{Q} + \omega_o^2}{s^2 + s\dfrac{\omega_o}{KQ} + \omega_o^2}$

Perceptual Threshold : subjectively detectable difference between two signals.

Phase Delay : ratio of the phase to the frequency.

$$\tau_p = -\frac{\phi}{\omega}$$

Pot : potentiometer.

Q: quality factor. In the case of a second-order bandpass filter

$$Q = \frac{\omega_0}{\omega_2 - \omega_1}$$

where ω_1 and ω_2 correspond to the lower and upper -3 dB frequency points respectively on each side of the center frequency ω_o. In the case of a second-order equalizing filter, Q corresponds to the parameter Q in

$$H_{BE}(s) = \frac{s^2 + s\frac{A\omega_o}{Q} + \omega_o^2}{s^2 + s\frac{B\omega_o}{Q} + \omega_o^2}$$

j: $\sqrt{-1}$.

SSE : sum of squared errors.

θ : frequency dependent variable. In the case of a second-order filter:

$$\theta = Q\left[\frac{\omega}{\omega_o} - \frac{\omega_o}{\omega}\right]$$

In the case of a first-order shelving filter:

$$\theta = \frac{\omega}{\omega_o}$$

z^{-1} : unit delay operator.

Chapter 1

1. Introduction

1.1. Problem Statement

The goal of this study was to design a structure and an algorithm by means of which the magnitude frequency response of a loudspeaker inside a listening room could be equalized automatically.

1.2. Background

Sound system equalization is the process of adjusting the electronic frequency response to compensate for undesired loudspeaker response and room acoustics. In a playback system, the goal is to make the total system sound more natural and in a live performance it consists of making the system sound more natural as well as avoiding feedback problems caused by peaks in the system frequency response [1] [2] [3] [4].

The proper equalization of a sound system is a time consuming task and the system must be re-equalized whenever any parameter is changed. Traditionally, room equalization has been done with analog graphic and parametric equalizers [3] [5].

1.2.1. Graphic Equalizers

A graphic equalizer basically consists of an electronic unit with an input and an output with a series of slider potentiometers (pots) on the front panel. Each slider pot controls the amount of gain boost or cut in a specific frequency band. The amount of boost or cut is indicated by the vertical position of the slider and the affected frequencies depend on which slider is adjusted.

Typically, the center frequencies are spaced at octave intervals, half-octave or third-octave intervals with approximately ten, twenty and thirty bands respectively each covering the entire audio band with increasing amount of resolution.

Ideally the front panel settings would provide a graphical representation of the magnitude frequency response of the graphic equalizer - hence the name. However, this turns out to be one of the most difficult aspects in the design of analog and digital graphic equalizers.

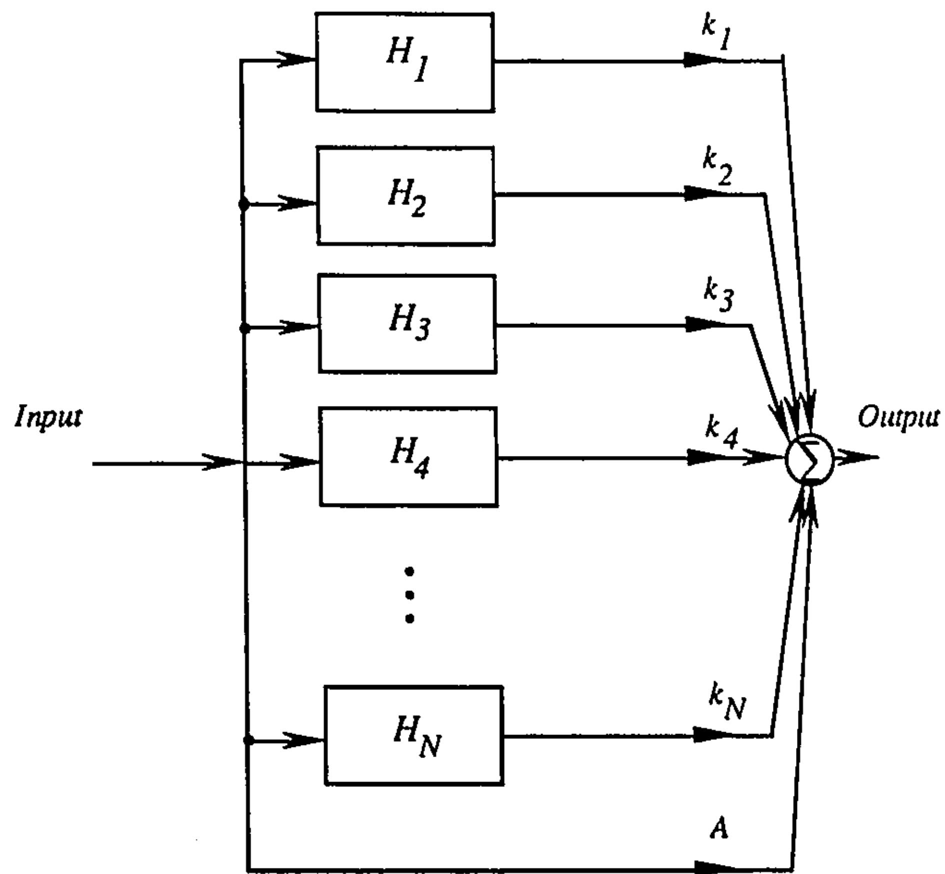


Figure 1.1 Analog Graphic Equalizer Block Diagram

1.2.1.1. Analog Graphic Equalizers

1.2.1.1.1. Topology

Although several topologies are possible for graphic equalizers only very few are used. Most have the topology shown in Figure 1.1 in its boost configuration. It consists of several bandpass filters whose outputs are summed and added to the input times a constant (usually one). The transfer function of this dominant topology [6][7] for the case of only one boost section is :

$$1 + k H_{bp}$$

where H_{bp} is a bandpass filter.

The main advantage of this type of configuration is that in the absence of any equalization, the input signal is unaltered. The output of each bandpass filter is zero and thus no noise or distortion is added. This would not be the case for a cascade topology for instance where the signal would need to go through each section, adding noise and distortion even when no equalization is performed. Another possible topology is one similar to Figure 1.1 but without the input path to the output summer. In this case the output weights are set to 1 for no equalization. This topology has the disadvantage that in the absence of any equalization, noise and distortion is added.

For a cut, the transfer function could be of the form :

$$1 - k H_{bp}$$

However, this results in unsymmetrical boost and cut curves as shown in Figure 1.2 for the case of a center frequency of 750 Hz and a boost and cut of 12 dB. The boost curve is not a mirror image of the cut curve reflected through the 0 dB line even though all the parameters as the same. Pole-zero plots are shown in Figure 1.3 for one stage of such an equalizer. It is seen that the pole positions are fixed while the zeros move away from the imaginary axis behind the poles for a boost and towards the imaginary axis in front of the poles for a cut. The positions of the poles and zeros are a function of the bandpass

function used. This issue will be discussed in section "Bandpass Filter Order and Q".

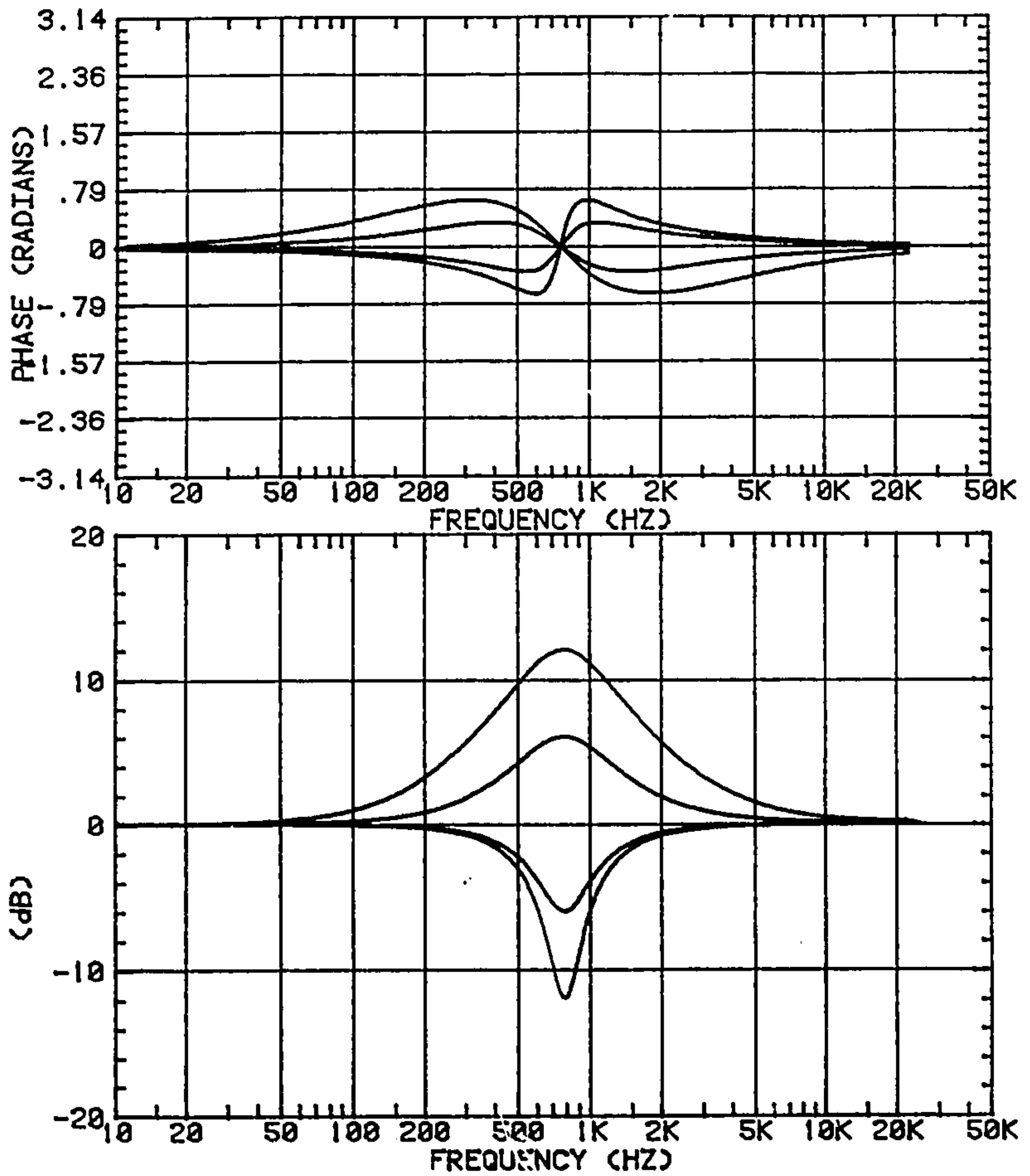


Figure 1.2 Unsymmetrical Boost and Cut Equalizer Response

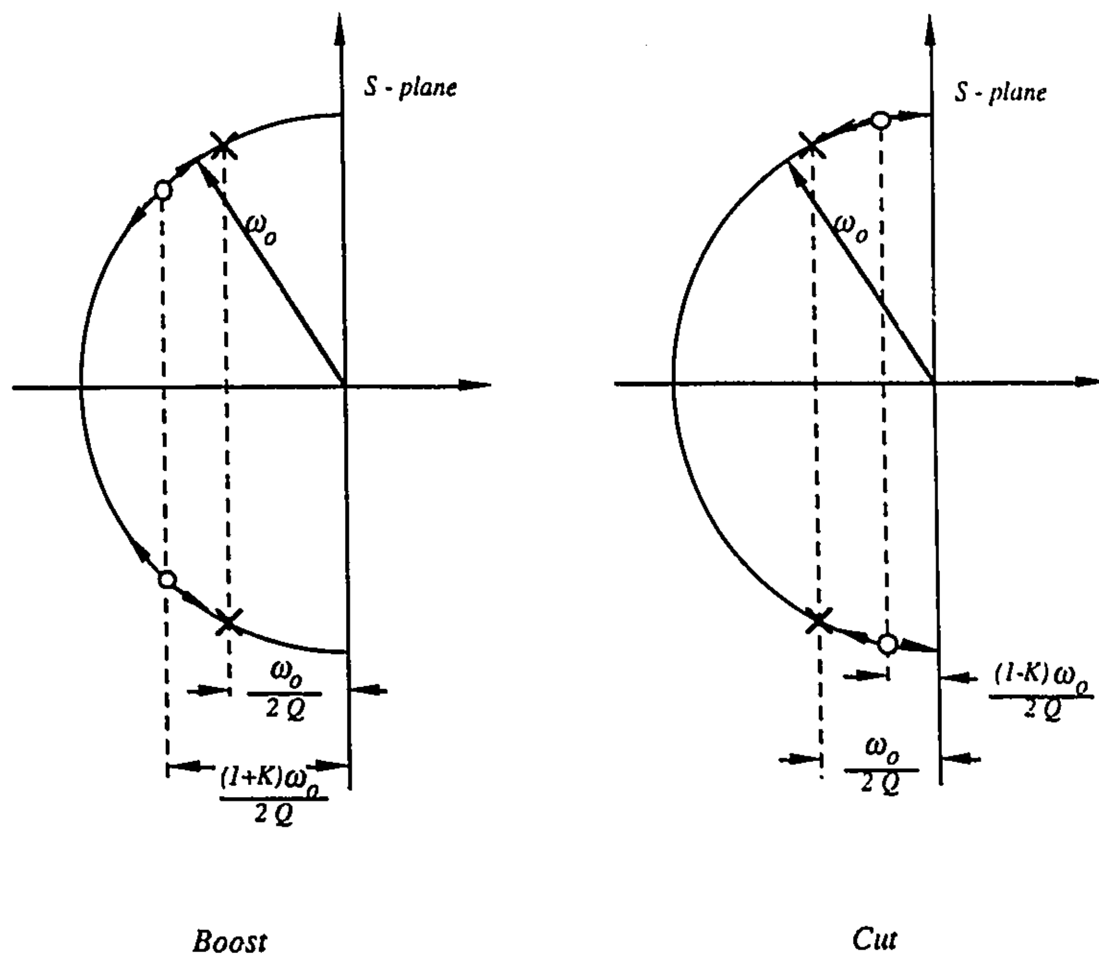


Figure 1.3 Unsymmetrical Boost and Cut Pole-Zero Locations

1.2.1.1.2. Boost and Cut Symmetry

Audio engineers prefer symmetrical boost and cut curves [6]. In order to achieve that, the following transfer functions are used :

$$Boost : 1 + kH_{bp} \quad (1.1)$$

$$Cut \; : 1 / (1 + kH_{bp}) \quad (1.2)$$

This is accomplished using the circuit of Figure 1.4.

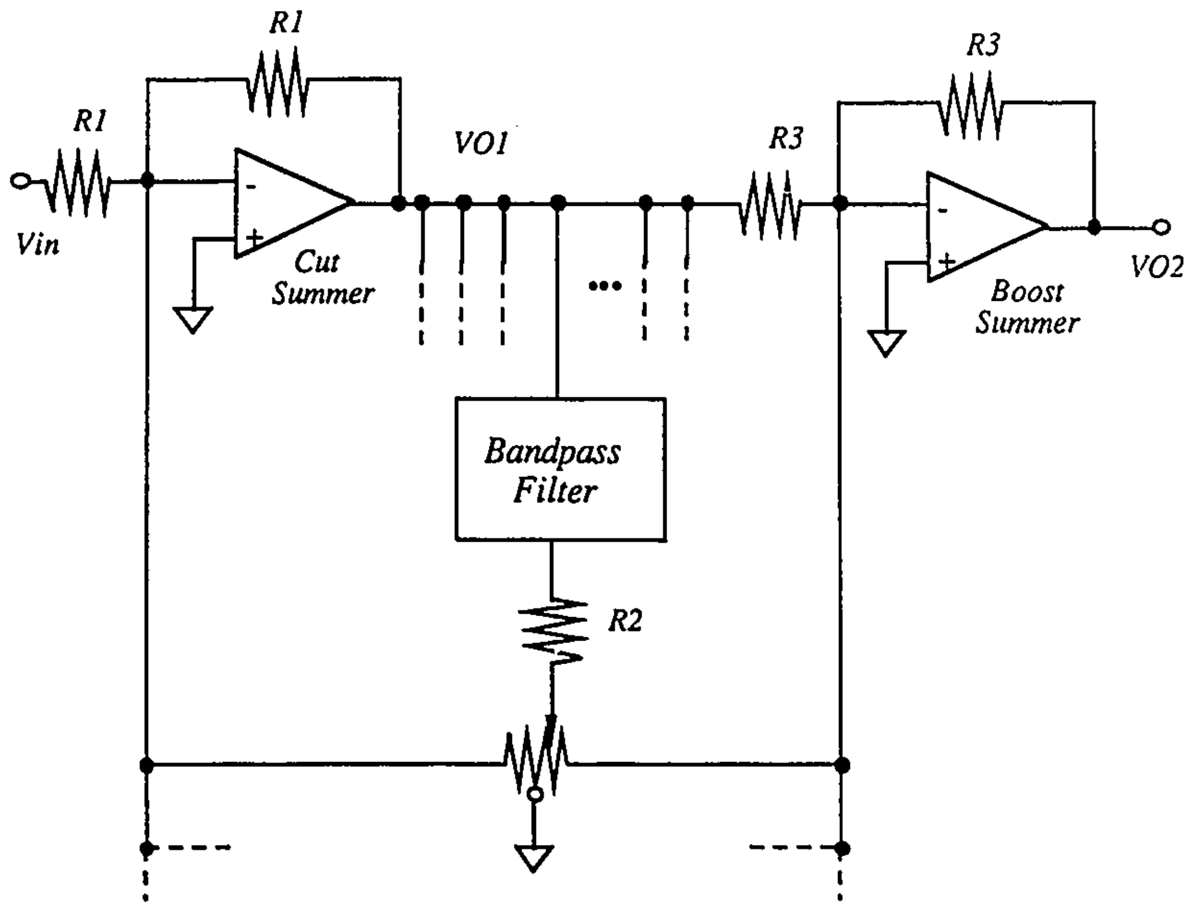
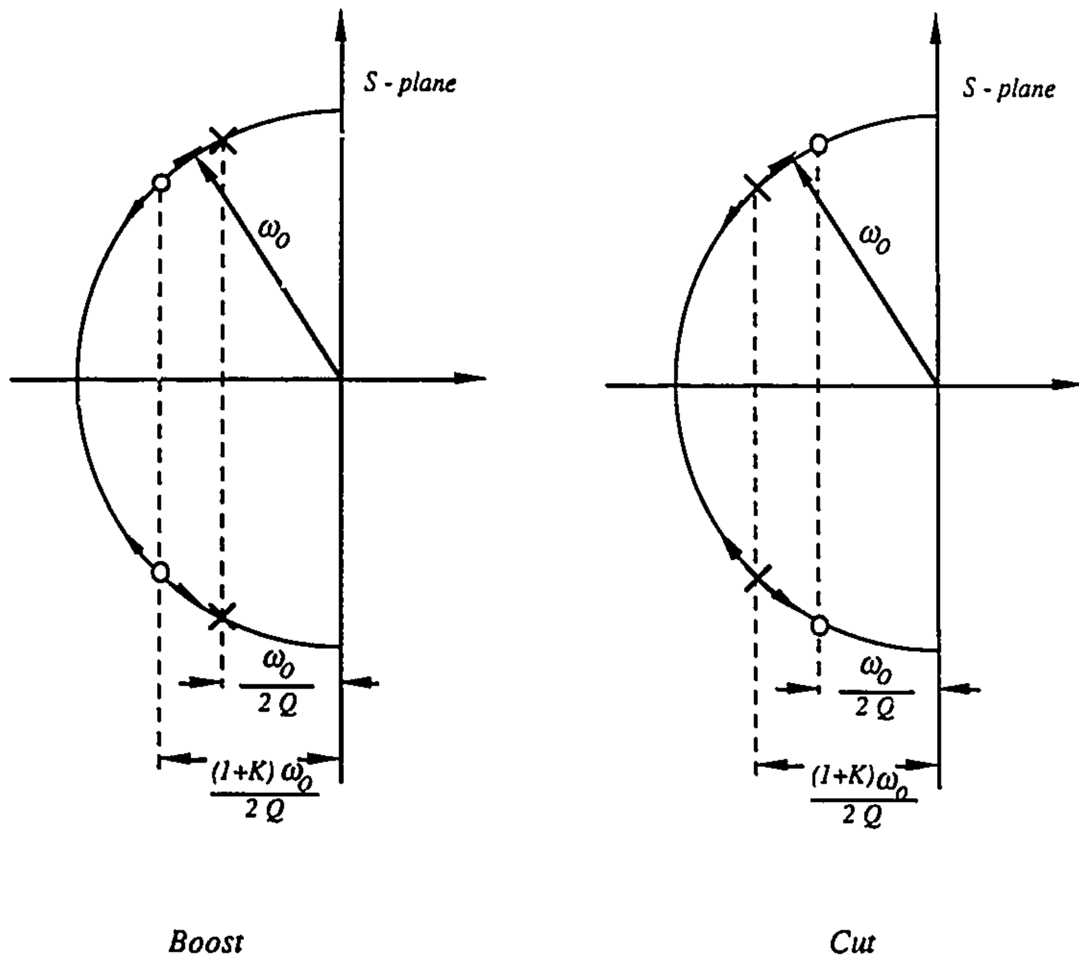


Figure 1.4 Symmetrical Boost and Cut Graphic Equalizer Schematic

The frequency response for this type of equalizer is shown in Figure 1.6 for the case of 12 dB and 6 dB boost and cut with a Q of 1 and center frequency of 750 Hz. As seen from this figure the curves are symmetrical. Notice also that the curves are not linear on a dB basis. At 750 Hz, the gain is 6 dB and 12 dB for the two boost curves. At 200 Hz, the gain is 0.9 dB and 3.3 dB respectively. Therefore, when the gain is changed at a given center frequency, the gain at

adjacent frequencies is changed in a non-linear fashion on a dB basis. The issue of gain linearity on a dB basis is addressed in Chapter 3 and an attempt at solving interband interference is described in Chapter 2.

Pole-zero plots are shown in Figure 1.5 for one stage of the equalizer of Figure 1.4. In the case of a boost, the poles are fixed and the zeros move away from the imaginary axis behind the poles. In the case of a cut, the opposite happens: the zeros are fixed and the poles move away from the imaginary axis behind the zeros. The positions of the poles and zeros are a function of the bandpass function used. This issue will be discussed in the following section.



Boost *Cut*

Figure 1.5 Symmetrical Boost and Cut Pole-Zero Locations

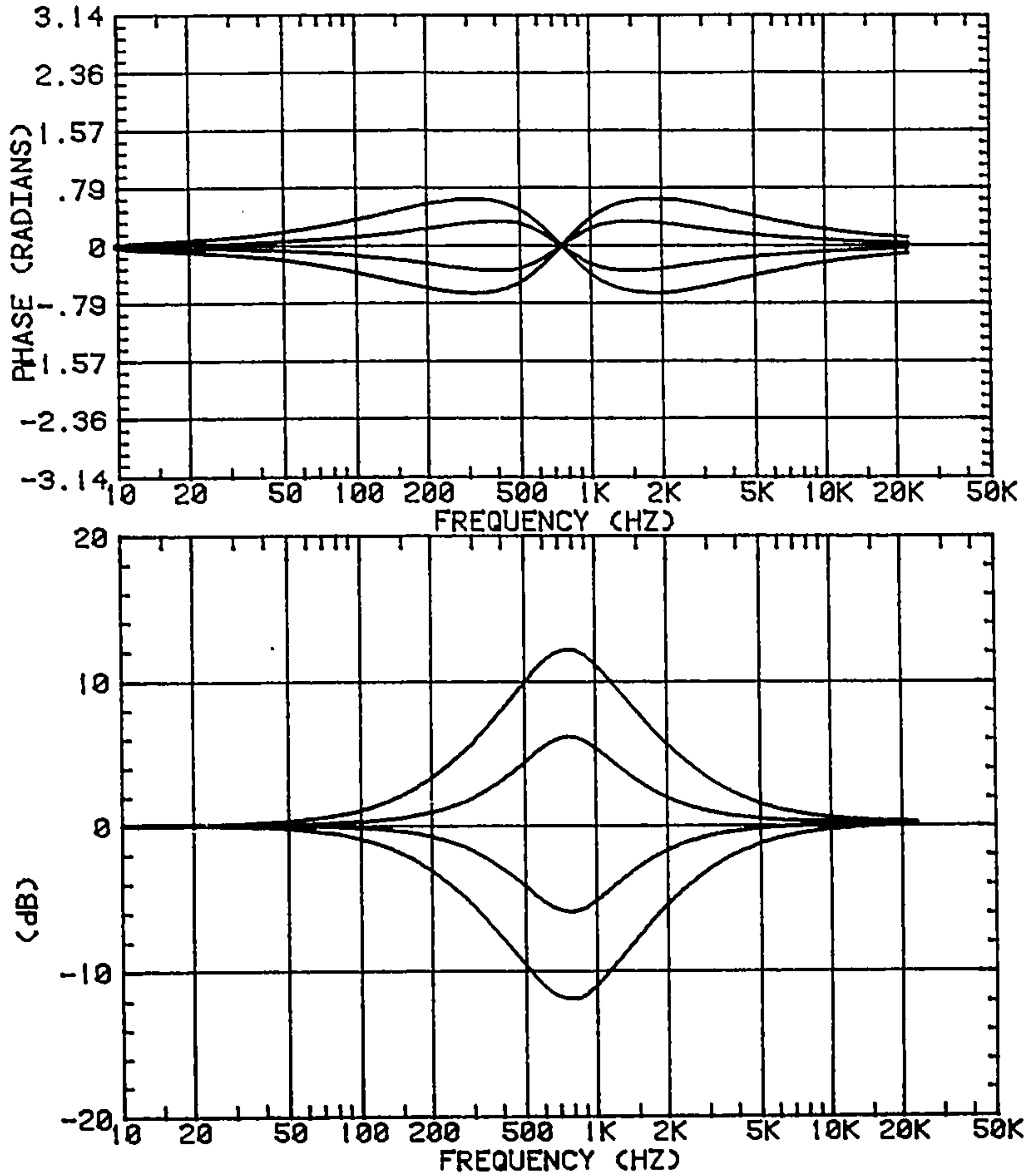


Figure 1.6 Symmetrical Boost and Cut Equalizer Response

1.2.1.1.2. Bandpass Filter Order and Q

It is common among graphic equalizer designs to use a second-order bandpass function of the form

$$H_{bp}(s) = \frac{s\frac{\omega_o}{Q}}{s^2 + s\frac{\omega_o}{Q} + \omega_o^2} \qquad (1.3)$$

By substituting $s=j\omega$ in (1.3), taking the magnitude and setting the result equal to $1/\sqrt{2}$ it is found that

$$Q = \frac{\omega_o}{\omega_2 - \omega_1} \qquad (1.4)$$

where ω_1 and ω_2 correspond to the lower and upper -3 dB frequency points respectively on each side of the center frequency ω_o. Q is the quality factor and is a measure of the sharpness of the filter. $B = \omega_2 - \omega_1$ is known as the -3 dB bandwidth. Octave graphic equalizers with center frequencies an octave apart and with Q value such that the -3 dB bandwidth is an octave wide are known as "true octave equalizers" [7]. Similarly, true third-octave equalizers have center frequencies one third of an octave apart and each bandpass filter has a -3 dB bandwidth one third of an octave wide. Some types of third-octave graphic equalizers have center frequencies spaced one third of an octave apart but the -3 dB bandwidths are not one third of an octave wide. They have lower Q's and exhibit significant interband interference. For instance, if one slider is set for say 3 dB boost at some center frequency f_o and all others are set

to zero then the boost at f_0 is indeed 3 dB. However, if the two adjacent sliders are set to 3 dB as well, then the gain at f_0 will be significantly higher than 3 dB. This is due to the significant overlap of the transition bands of adjacent filters. A solution to this problem would be to use higher Q filters or to increase the order of the filters. This would reduce the interaction between the bands but if the Q or the order is too high, objectionable ripple in the magnitude frequency response is introduced. Therefore a compromise must be reached between interband interference and frequency response smoothness [6] [7] [8].

Early designs had both problems. They had significant interband interference at low boost or cut setting while having little at high boost or cut setting but significant ripple. Early designs used capacitors, inductors and variable resistors and a topology different than Figure 1.4. These designs were such that the resistors varied the amount of boost or cut but in addition, the Q of the circuit varied as well. The higher the boost or cut the higher was the Q. At small amount of boost or cut, the value of Q was low such that the interaction between bands was significant. Constant-Q designs would have required extra circuitry. Later designs used gyrators to simulate inductors but these were still not constant Q. In recent years, constant-Q designs have emerged since they can now be economically manufactured using integrated circuit technology [7] [9]. The advantage of constant-Q designs is that the Q value can be optimized to get a better compromise between interband interference and magnitude ripple compared to traditional non-constant-Q designs [6]. However, it should be pointed out that these effects are not eliminated.

True octave equalizers have bandpass filters with Q's of 1.4 and true 1/3-octave equalizers have Q's of 4.318. These numbers are obtained by substituting

$$\omega_1 = \frac{\omega_0}{2^{1/2}}, \quad \omega_2 = 2^{1/2}\,\omega_0 \text{ for octave equalizers}$$

and

$$\omega_1 = \frac{\omega_0}{2^{1/6}}, \quad \omega_2 = 2^{1/6}\,\omega_0 \text{ for third octave equalizers}$$

in equation (1.4). Graphic equalizers use second-order sections instead of higher-order ones seemingly for economic reasons. Bohn [6] studied the impact of filter order on the frequency response of graphic equalizers and found that second-order sections are actually optimum in the sense that adjacent bands combine at the -3 dB points to unity and zero degree phase shift. At the -3 dB point of two adjacent filters, the magnitude response is $1/\sqrt{2}$ with a phase of +45 degrees for the upper filter and a magnitude of $1/\sqrt{2}$ and a phase of -45 degrees for the lower summing to 1 and zero phase. The use of higher-order bandpass filters in a graphic equalizer results in excessive ripple in the magnitude response.

1.2.1.1.3 Phase Response

The frequency response of a system can be characterized as

$$H(\omega) = A(\omega)\, e^{j\phi(\omega)} \tag{1.5}$$

where

$A(\omega)$ is the magnitude response

and

$\phi(\omega)$ is the phase response.

The system can also be described using a logarithmic function

$$\ln H(\omega) = a(\omega) + j\phi(\omega) \tag{1.6}$$

where $a(\omega) = \ln A(\omega)$. Therefore, the real part of (1.6) is the magnitude response in Nepers and the imaginary part is the phase response in radians.

A minimum-phase system is one whose phase and log-magnitude response form a Hilbert transform pair [10]:

$$a(\omega) = \frac{1}{\pi} \, P \int_{-\infty}^{\infty} \frac{\phi(\omega)}{\omega - x} \, dx \tag{1.7}$$

$$-\phi(\omega) = \frac{1}{\pi} \, P \int_{-\infty}^{\infty} \frac{a(\omega)}{\omega - x} \, dx \tag{1.8}$$

where P indicates that the principal part of the integral is to be taken at the pole ω. Therefore the phase and magnitude of a minimum-phase system are inextricably related. One cannot be changed without the other. Such a system has the property that all its poles and zeros lie in the left-hand side of the s-plane. To be stable, a system must have all its poles in the left-hand side of the s-plane. However, the zeros can be on the right side and the system is still stable. If a system has at least one zero on the right side of the s-plane, such a system is said to be non-minimum phase. Greiner and Schoessow [7] analyzed the three most common graphic equalizer topologies and concluded that they were all minimum phase. Bohn [6] went so far as to state that

> *"In fact, no examples of commercially available graphic equalizers exist that are not minimum phase".*

Discrete time versions of (1.7) and (1.8) exist and the reader is referred to [11]. A stable minimum-phase digital system is one whose poles and zeros lie inside the unit circle ($z = 1$). A non-minimum-phase digital system has at least one zero outside the unit circle.

1.2.1.2. Digital Graphic Equalizers

Recently, with the advent of digital signal processing (DSP) technology, better control over the equalization process is possible in addition to the capability of automating the adjustment procedure. Different approaches are now taken in the use of this new DSP technology as it applies to sound equalization. One approach is to duplicate the old analog functionality and implement it in DSP for increased precision and control. More specifically,

these equalizers attempt to modify only the magnitude response of the system without attention to the phase or time domain parameters such as reflections. This first approach falls into two categories : FIR designs [8] [12] [13] [14] [15] [16] [17] and IIR designs [18] [19] . The same trade-offs usually associated with FIR and IIR filters apply here: IIR filters are more efficient, but are limited to non-linear phase behavior and are difficult to design. FIR filters are less efficient but allow linear phase to be achieved and are easier to design. However, in order to make the FIR equalizer realization using typical computing capability of DSP processors, even many in parallel, the FIR equalizer must be designed using decimation techniques. This greatly improves the computing efficiency but has the drawback of introducing a fixed delay. This limits the application of FIR equalizers to playback only (and not live performances) [8].

Filterbanks fall into the category of digital graphic equalizers since they can be made to perform this function. Traditionally however, they have been used mostly in speech analysis, synthesis and coding [20].

1.2.2. Parametric Equalizers

Graphic equalizers, as explained earlier, consist of several bandpass filters at fixed frequencies and fixed Q (or at least uncontrollable Q). Parametric equalizers on the other hand are more versatile since they consist of several bandpass filter sections with adjustable Q, center frequency and gain. They usually have fewer sections than graphic equalizers due to their added flexibility combined with the fact that generally only a portion of the audio band needs equalization. For instance, several sections can be used to equalize

one particular frequency region. This is not possible with a graphic equalizer since the center frequencies are fixed and therefore several filters often end up not being used. In another instance, a broad (low Q) bump in the magnitude frequency response may be equalized with only one parametric stage but would require many graphic equalizer stages.

The price to pay for this added flexibility however is the added complexity in adjusting the parametric equalizer since it is more difficult to form an accurate picture of the frequency response of the equalizer from the front panel settings. One almost always needs some means of measuring the effect of the parametric equalizer. Another disadvantage of the parametric equalizer is the complexity of the automation procedure. In the case of a graphic equalizer, the automation procedure would consist merely of adjusting the gain parameter at each center frequency whereas three parameters need to be adjusted in the case of a parametric equalizer (fo, Q and gain).

1.2.2.1. Analog Parametric Equalizers

The schematic diagram of most analog parametric equalizers is very similar to that of a graphic equalizer as shown in Figure 1.7. It consists of a subtracter and an adder circuit with several parallel filters in between. The function implemented for one equalizing section is still $1 + kH_{bp}$ for a boost and $1/(1 + kH_{bp})$ for a cut . The difference lies in the filter (H_{bp}). In a parametric equalizer, the center frequency and the Q of the filters can be adjusted in addition to the amount of boost or cut. This is usually achieved using state-variable filters [6] [46]. Analog components limit the range in which the

center frequencies can be varied. Typically each filter spans about three octaves.

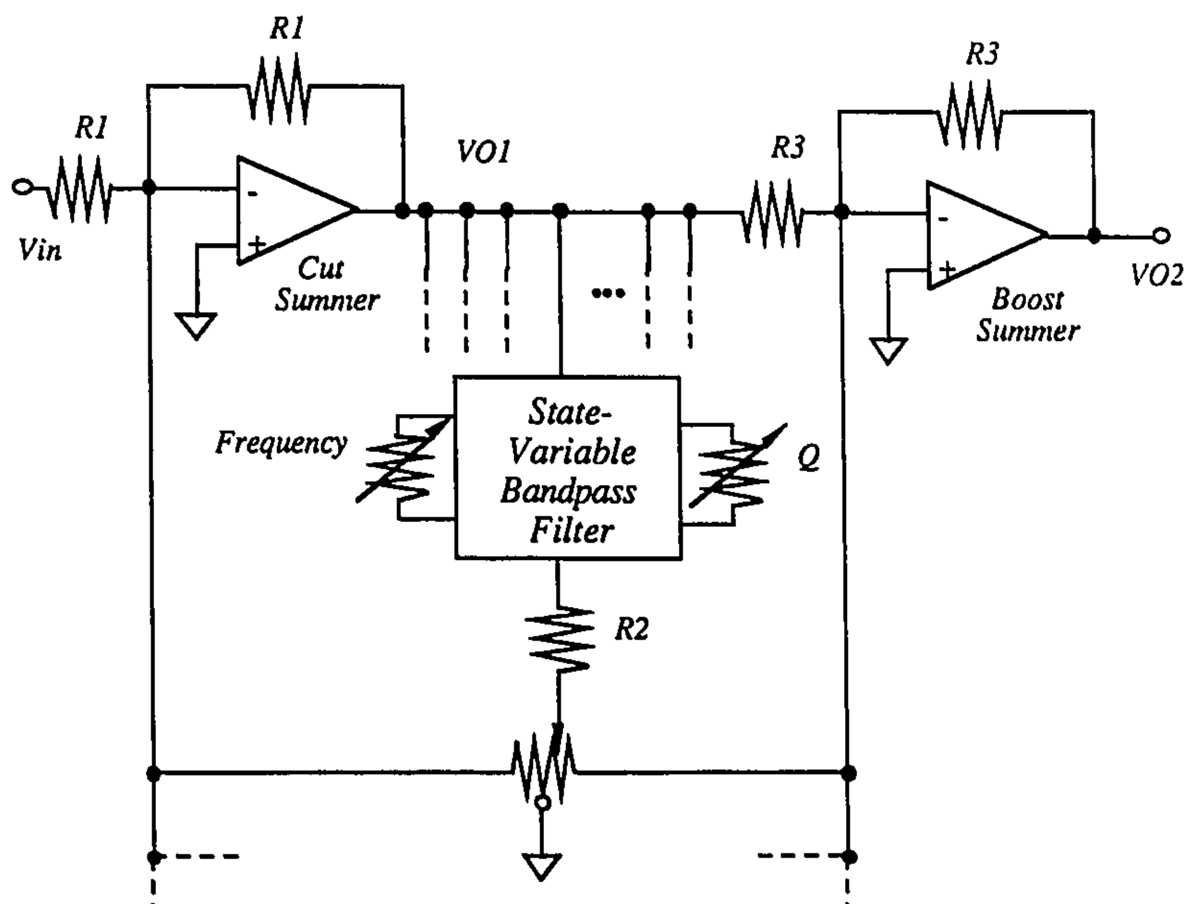


Figure 1.7 Analog Parametric Equalizer Schematic

1.2.2.2. Digital Parametric Equalizers

As far as we know, stand-alone digital parametric equalizers are not available on the market yet. The only references in the literature pertain to digital filters as applied to parametric equalizers [22] [23] [24]. No description of a complete stand-alone digital parametric equalizer was found.

1.2.3. Digital Filter Design Techniques

This is the case where digital IIR and FIR filters are designed to match arbitrary frequency responses or log-magnitude frequency responses. The main disadvantage with this method is that all the filter coefficients must be recalculated every time a single frequency point is changed. This method is not well suited for interactive equalization.

1.2.4. Adaptive Filters

This approach consists of single or multi-point equalization using digital adaptive filters with various types of adaptive schemes like LMS or least-squares [25] [26] [27]. It allows the possibility of equalizing directly in the time domain and theoretically compensate for such phenomena as wall reflections, time of arrival problems as well as magnitude. In general this method consists of designing a filter such that the impulse response of the total equalized system approximates some desired impulse response.

The filter type is generally FIR since it is easier to adapt than IIR but at least one reference has used a combination of both [26]. However, these systems are not without problems. The response of a room is usually non-minimum phase such that to perfectly invert the system requires acausal filters [26] [28].

1.3. Scope

In Chapter 2, a digital filterbank with allpass reconstruction property is investigated as a potential candidate for a graphic equalizer. The filterbank is analyzed from both an objective and a subjective point of view. The filterbank was found to effect a timbral modification of some musical signals even though the magnitude response was unity. For this reason, the filterbank was rejected as a candidate for a high-fidelity graphic equalizer.

Chapter 3 investigates two forms of parametric equalizers. The first uses results obtained by Bode [41] [43] and the second uses a modified-Bode approach which, as far as we know, is new. Two automation procedures are outlined. The first one is based on the work from Bullis and Brubaker [46] but failed to automatically optimize the parametric equalizer. The second procedure is iterative and succeeded. Simulations are presented where loudspeaker-room frequency responses were automatically equalized.

Finally conclusions are drawn and recommendations are made for further work.

Chapter 2

2. Allpass Filterbank

The merits of analog magnitude equalization of loudspeaker-room frequency response are well established [1] - [7]. However, the adjustment procedure is still very tedious. Since the types of analog filters used in these equalizers are inherently IIR, it was decided that a digital graphic equalizer using IIR filters would be useful. Such an equalizer would automatically adjust itself such that the cascade of the measurement frequency response and the equalizer would approximate some target frequency response. This equalizer would benefit from all the advantages that digital filters have over analog filters, namely: lower sensitivity, better programmability, better reliability, freedom from component drift and no matching of components required.

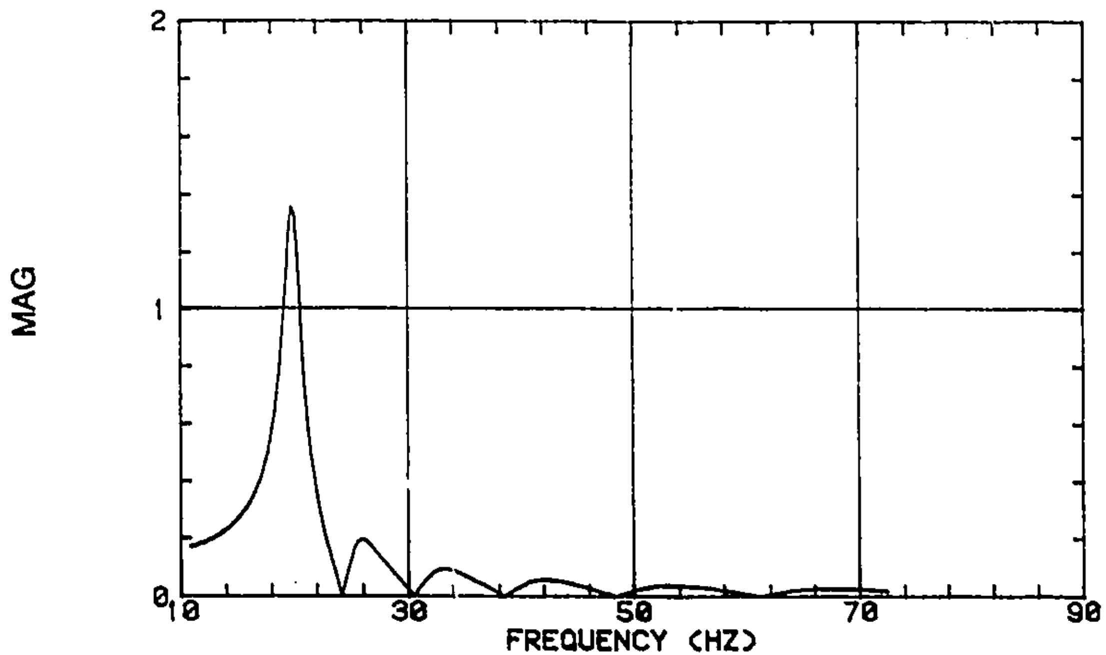


Figure 2.1 Filterbank Magnitude Response of one Stage

A new type of filterbank with frequency interpolating property has recently been introduced [18] [29] [30] [31] [32]. It consists basically of several filters centered at arbitrary frequencies. Each filter has the property that it has unity magnitude response at its center frequency and zero response at the center frequency of all others. This property is shown in figure 2.1 where the magnitude response for one filter is plotted. Note that the center frequency in Figure 2.1 is not at the maximum peak response but at a slightly lower frequency where the response is unity. The response of adjacent filters would add out of phase at the peak response such that the total system magnitude response would be unity as shown in Figure 2.3. The property that each filter has zero response at the center frequency of all other filters was initially deemed to be very desirable since for any gain setting, the response at the

center frequencies would be exactly as indicated by the weight term at those frequencies.

The filterbank can be described mathematically as follows:

$$H_{ap1}(z) = \frac{\sum_{k=0}^{N-1} \frac{w_k r_k z^{-1}}{1 - pp_k z^{-1}}}{1 + \sum_{i=0}^{N-1} \frac{pp_i r_i z^{-1}}{1 - pp_i z^{-1}}} \tag{2.1}$$

where

$pp_k = exp(j2\pi f_k)$

f_k = filter center frequency

r_k = real weights such that $\sum_{k=0}^{N-1} r_k = 1$

w_k = output weights

$j = \sqrt{-1}$.

As applied in this study, complex output weights are used and filters at $-f_k$ are required [18]. This yields the following filterbank equation:

$$H_{ap2}(z) = \frac{\sum_{k=0}^{N-1} \frac{w_k r_k z^{-1}}{1 - pp_k z^{-1}} + \frac{w_k^* r_k z^{-1}}{1 - pn_k z^{-1}}}{1 + \sum_{i=0}^{N-1} \frac{pp_i r_i z^{-1}}{1 - pp_i z^{-1}} + \frac{pn_i r_i z^{-1}}{1 - pn_i z^{-1}}} \tag{2.2}$$

where pp_k, f_k, r_k, w_k and j are defined as in (2.1) and

$pn_k = exp(-j2\pi f_k)$

w_k^* = complex conjugate of w_k.

Equation (2.2) is shown in a block diagram form in Figure 2.2.

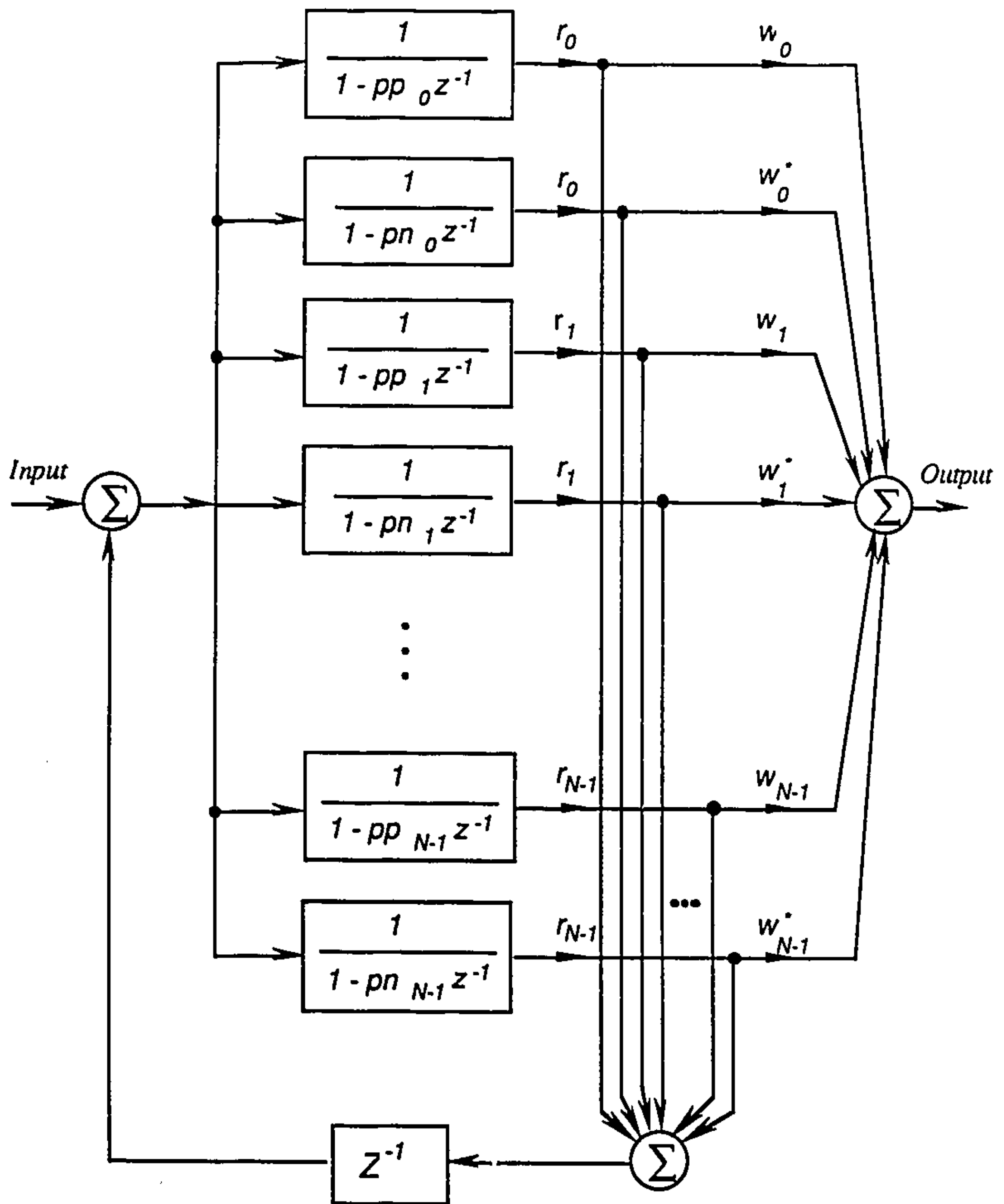


Figure 2.2 Filterbank Block Diagram

2.1. Analytic Frequency Response

The filterbank frequency response is easily found by substituting $z = e^{j\omega}$ in equation (2.2). The magnitude and phase responses are defined respectively as

$$Mag = 20 \log \left| H_{ap2}(e^{j\omega}) \right| \text{ in dB} \tag{2.3}$$

and

$$Phase = \tan^{-1}\left[\frac{Im\,(H_{ap2}(e^{j\omega}))}{Re(H_{ap2}(e^{j\omega}))}\right] \tag{2.4}$$

Equations (2.3) and (2.4) were implemented in a C program listed in Appendix A. The resulting frequency response is shown in Figure 2.3 with coefficients as listed in Table 2.1. Notice that the magnitude is exactly unity (0 dB) and that the phase is non-linear. The unwrapped phase is shown in Figure 2.4. Figure 2.5 shows the filterbank frequency response for the case where all output weights have unity magnitude except one set at 0.316 (-10 dB).

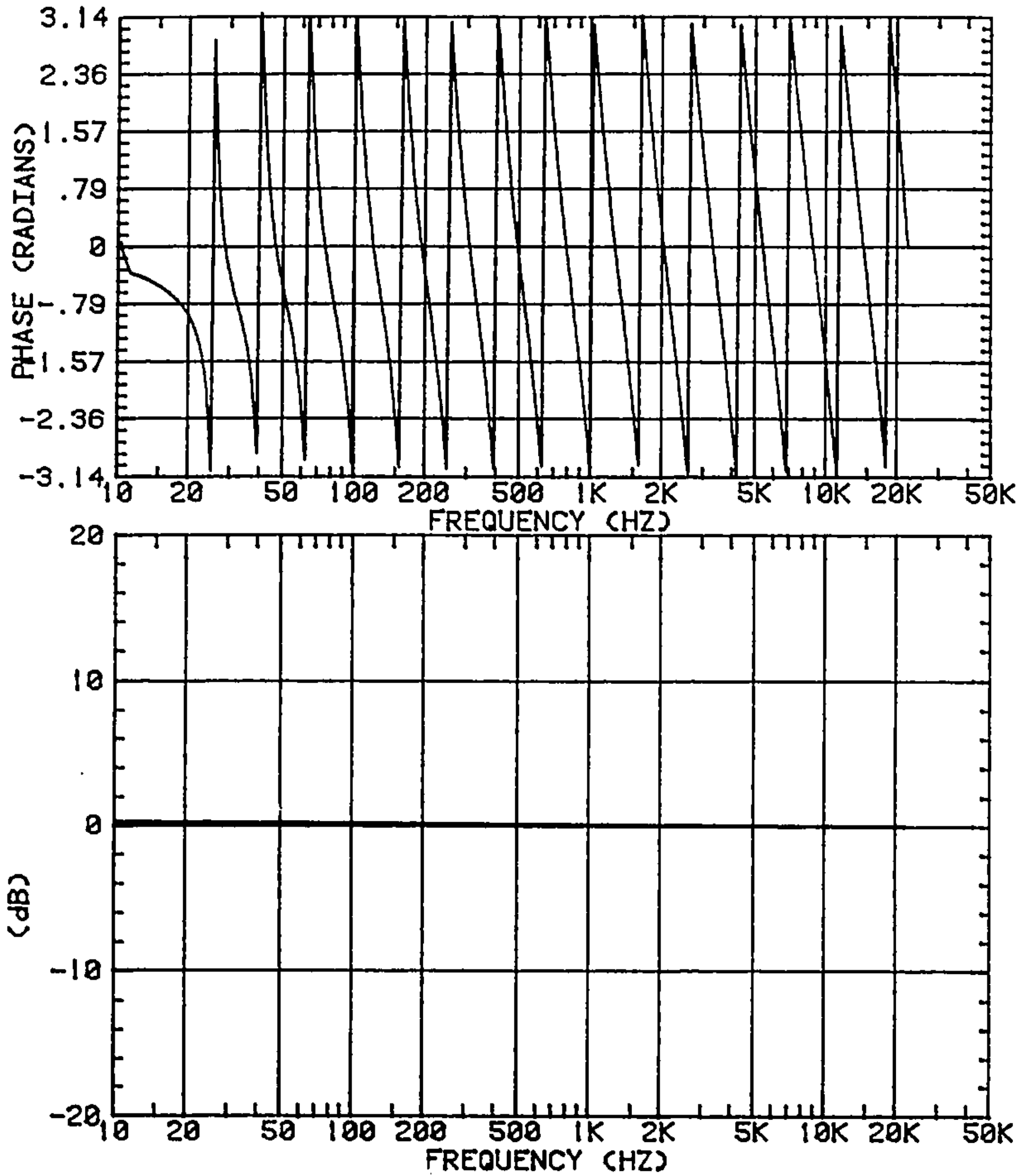


Figure 2.3 Allpass Filterbank Frequency Response

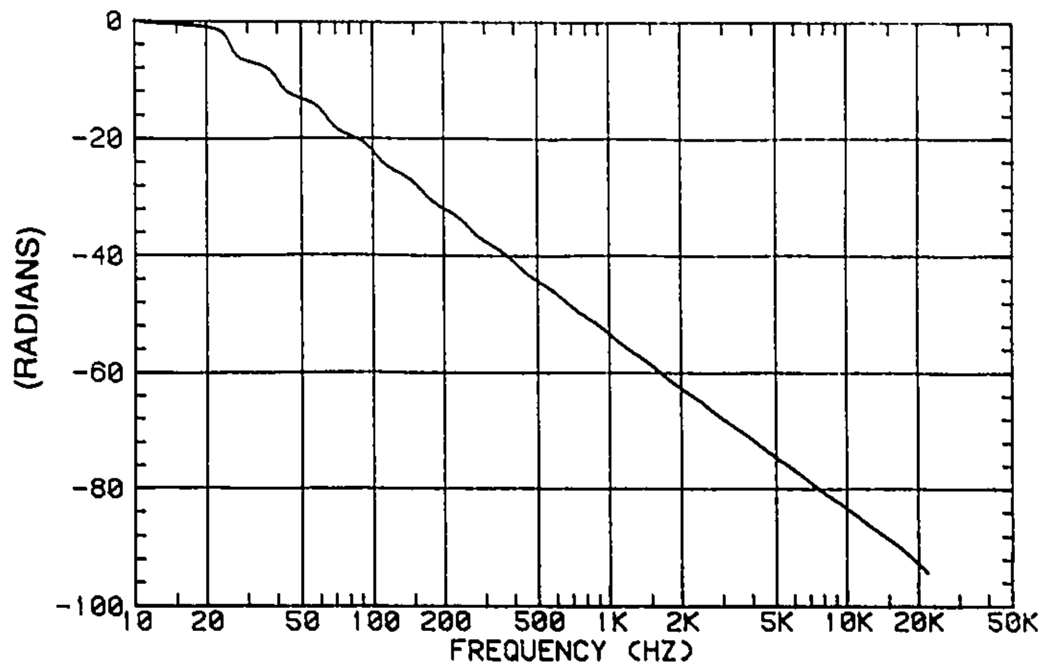


Figure 2.4 Unwrapped Phase Response

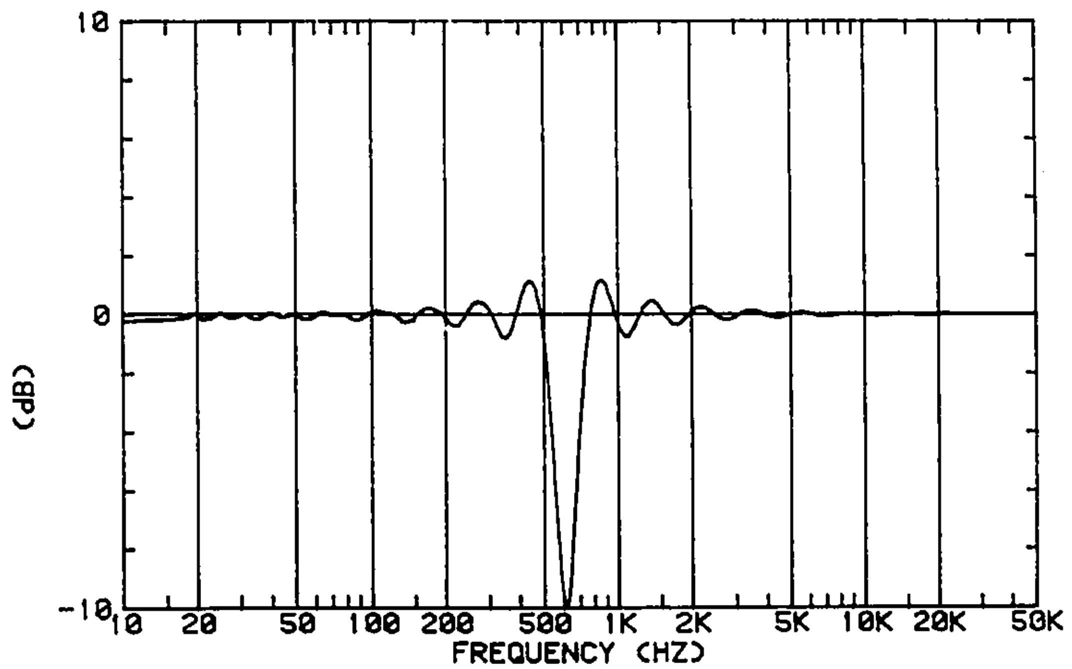


Figure 2.5 One Weight at -10 dB

Frequency (degrees)	r_k	angle of w_k (degrees)
160.361770	0.1032297502	104.989000
127.279221	0.0819335070	-61.667080
101.021585	0.0650306676	99.943930
80.180885	0.0516148751	-82.034260
63.639611	0.0409667535	77.685930
50.510792	0.0325153338	-104.520810
40.090442	0.0258074375	55.919880
31.819805	0.0204833768	-124.641650
25.255396	0.0162576669	37.024280
20.045221	0.0129037188	-140.656880
15.909903	0.0102416884	22.255930
12.627698	0.0081288334	-151.890990
10.022611	0.0064518594	11.512690
7.954951	0.0051208442	-158.920770
6.313849	0.0040644167	3.794500
5.011305	0.0032259297	-162.760660
3.977476	0.0025604221	-2.005590
3.156925	0.0020322084	-164.278860
2.505653	0.0016129648	-6.780530
1.988738	0.0012802110	-164.056690
1.578462	0.0010161042	-11.197040
1.252826	0.0008064824	-162.423340
0.994369	0.0006401055	-15.771700
0.789231	0.0005080521	-159.512760
0.626413	0.0004032412	-20.966130
0.497184	0.0003200528	-155.287920
0.394616	0.0002540260	-27.339980
0.313207	0.0002016206	-149.477480
0.248592	0.0001600264	-36.065890
0.197308	0.0001270130	-141.203380
0.156603	0.0001008103	-53.732480

Table 2.1 All-Pass Filterbank Coefficients

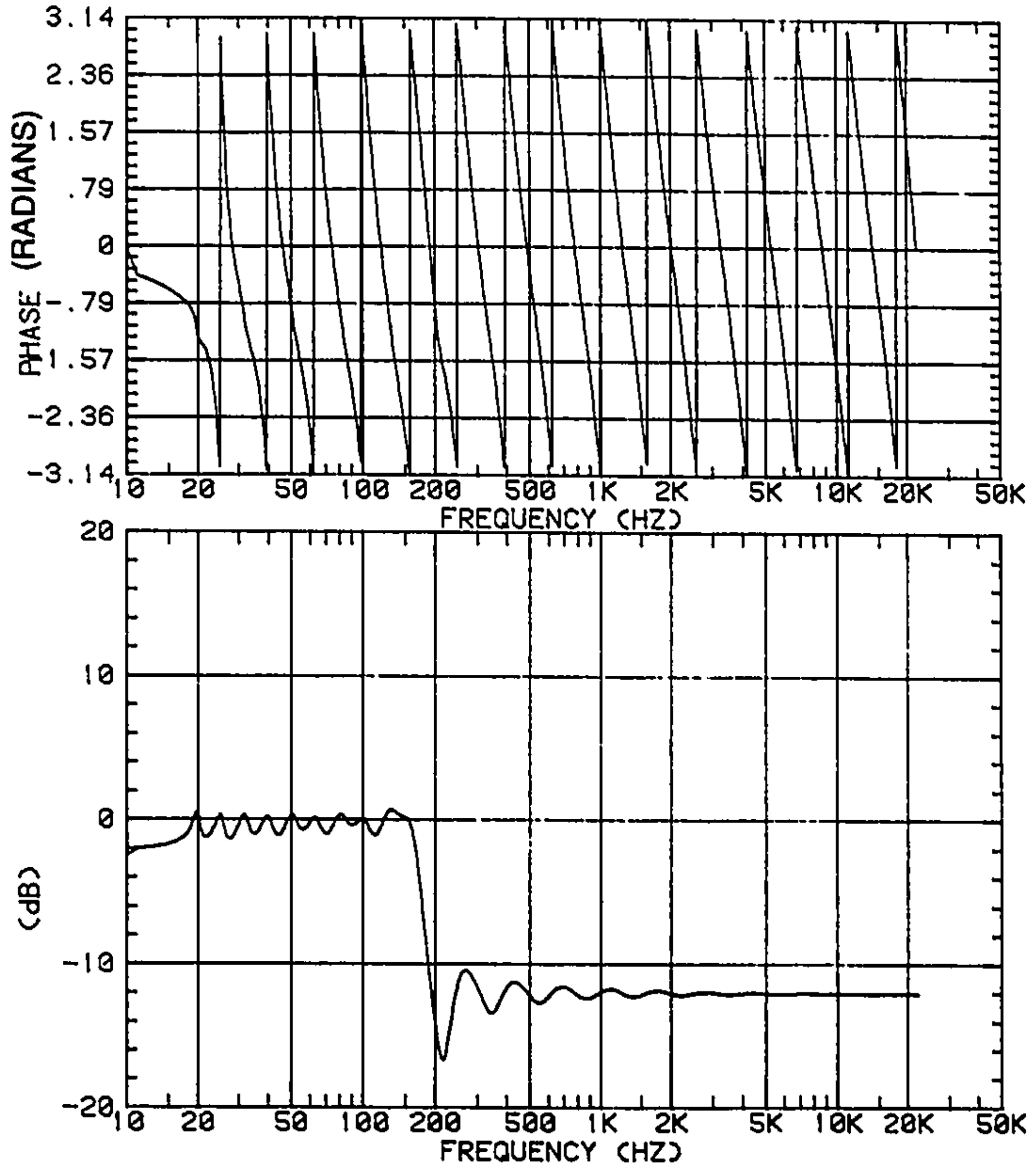


Figure 2.6 Frequency Response with High-Frequency Weights at -12 dB

Figure 2.6 shows the frequency response of the filterbank (eq. 2.2) with coefficients as listed in Table 2.1 but with output weight magnitudes for the lower ten frequencies set to unity and the remaining set to 0.25. Notice that the magnitude at each center frequency corresponds exactly to the output weight magnitude. There is however some ripple between the center frequencies.

2.1.2. Phase Response

Recall that any linear system can be represented by a magnitude $A(\omega)$ and a phase $\phi(\omega)$ as follows:

$$H(j\omega) = A(\omega)\, e^{j\phi(\omega)} \qquad (2.5)$$

For a distortionless system, it is necessary and sufficient that $A(\omega)$ be constant and $\phi(\omega)$ be linear in ω [33]. Thus the magnitude frequency response must be flat and the phase must be a straight line through the origin. The slope of the straight line represents a constant delay. When these conditions are not met, linear distortion (waveform distortion) is present [34] [35]. The filterbank defined by (2.2) is non-minimum phase with the position of the zeros alternating between inside and outside the unit circle. Therefore, even at the flat magnitude setting, the phase response is non-linear as shown in Figure 2.3 and linear distortion is introduced.

Two common measures of phase non-linearity are the phase delay

$$\tau_p = -\frac{\phi}{\omega} \tag{2.6}$$

and the group delay

$$\tau_g = -\frac{d\phi}{d\omega} \tag{2.7}$$

Phase delay can be interpreted as the amount of delay a steady-state sinusoid at some frequency would undergo through a system thus giving a measure of delay as a function of frequency. Group delay is the rate of change of the phase. Many studies relate group delay distortion to perceptual thresholds [34] [36] [37] [38]. It will therefore be interesting to relate our results to those studies. Some [10] [33] have argued that group delay is frequently not a meaningful measure of the time delay of a frequency component. Nonetheless, the filterbank group delay was calculated since it is a prevalent measure in the literature and perceptual thresholds for group delay have been investigated. Figure 2.7 shows the allpass filterbank phase delay with unity magnitude response. Notice how the low frequencies are delayed with respect to the high frequencies. Figure 2.8 shows the group delay with unity magnitude response. The filterbank has large group delay peaks at low frequency.

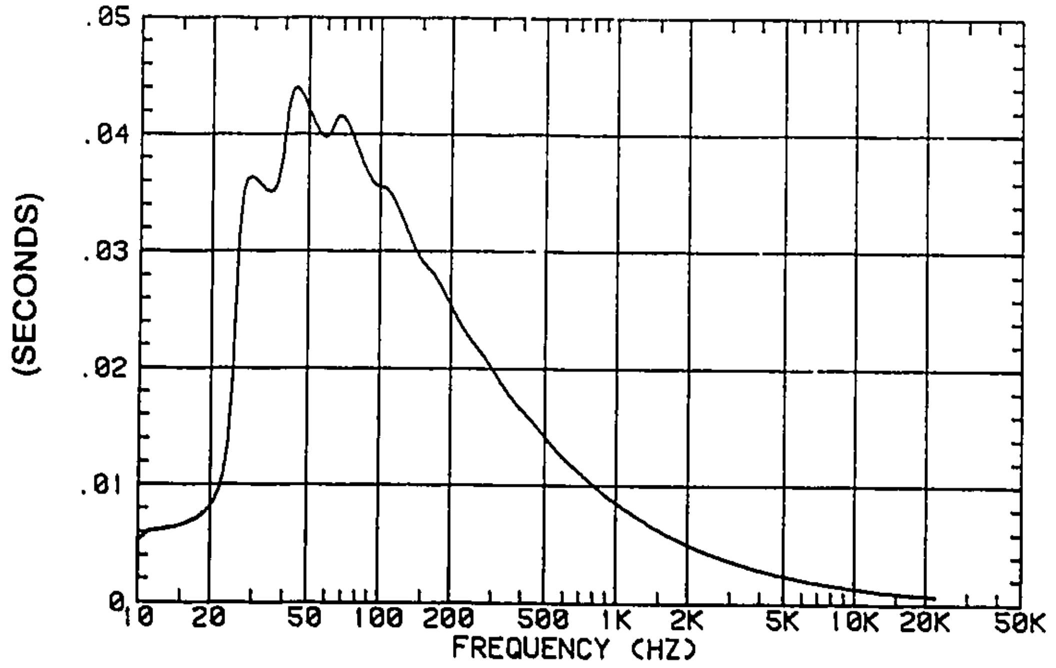


Figure 2.7 Allpass Filterbank Phase Delay

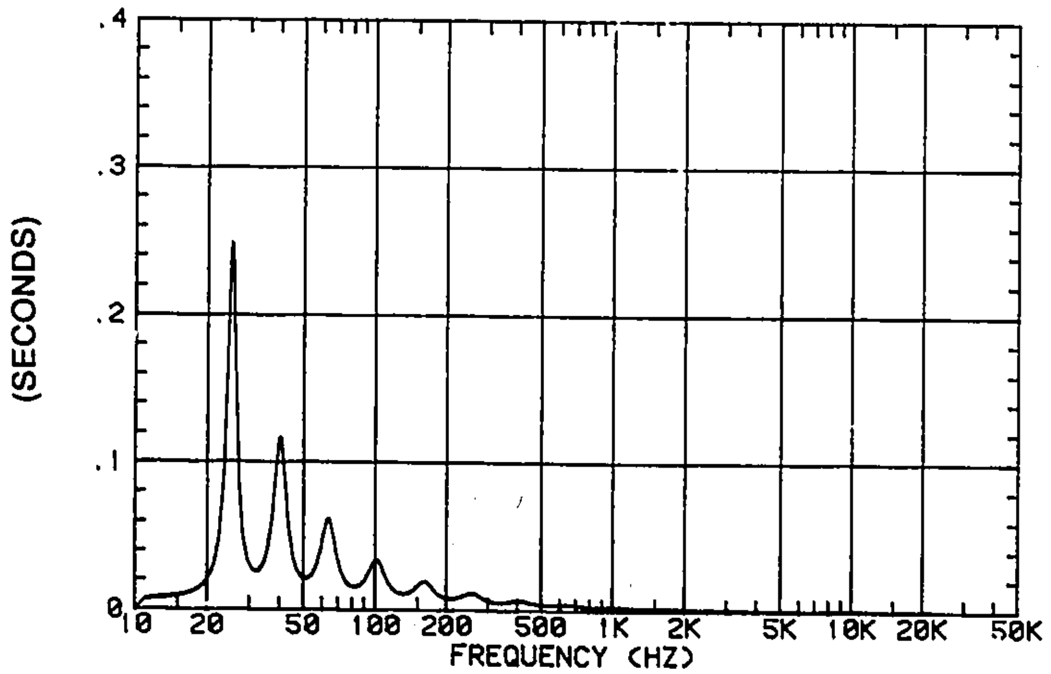


Figure 2.8 Allpass Filterbank Group Delay

When comparing the audibility of the filterbank group delay with the thresholds in the literature, one encounters two difficulties. First, the literature deals with the audibility of group delay distortion of one frequency component at a time. The filterbank has group delay distortion over the entire audio band. However, it seems reasonable to assume that if at least one frequency band exhibits group delay in excess of the threshold found in the literature, then it might be audible under the same listening conditions. The second difficulty is that the test signals are not the same. However, comparisons will be made to get an approximate assessment of the audibility of the filterbank group delay.

Preis [34] found that a group delay in excess of ±0.5 ms was audible in the 1 to 5 kHz range and a group delay in excess of ±2.5 ms was audible in the 100 to 400 Hz range. The filterbank under study had a group delay ranging from about 32 ms to about 8 ms for frequencies between 100 and 400 Hz and a group delay ranging from about 2.5 ms to about 0.4 ms for frequencies between 1 and 4 kHz. Therefore the low frequency threshold in Preis' study is greatly exceeded. The high frequency threshold is exceeded except for frequencies around 4 kHz.

A different version of the filterbank was tested. The parameters are listed in Table 2.2. The output weights had unity magnitude as in Table 2.1 but were real. Figure 2.9 shows the frequency response of this filterbank with coefficients as listed in Table 2.2 but with output weight magnitudes for the lower ten frequencies set to unity and the remaining set to 0.25 as in Figure 2.6 for the complex-output-weights case. Notice that the phase response oscillates around 0 radians and does not wrap around as in the case of the complex-

output-weights filterbank. Also notice that the price to pay for real output weights in this case is the loss of control over the magnitude response. The response does match the magnitude of the output weights at the center frequencies but the ripple in between is excessive.

It is interesting to note that interband independence at the center frequencies has been tried by Kraght [8] for a digital graphic equalizer and similar results were obtained even though the approach differed. The author first tried complete interband independence and found that unless all band amplitudes were the same, the summed frequency response had large ripples. Then this requirement was relaxed to interband independence at only the center frequencies. Again Kraght concluded that:

> *"However, it is impossible to have complete independence between the frequency bands and have a smooth frequency response at the same time."*

This conclusion led Kraght to a cubic-spline design which was a compromise between interband independence and frequency response smoothness. His design uses decimated FIR filters and therefore has a large delay which cannot be used for live performances. In addition, the coefficients cannot be updated in real-time.

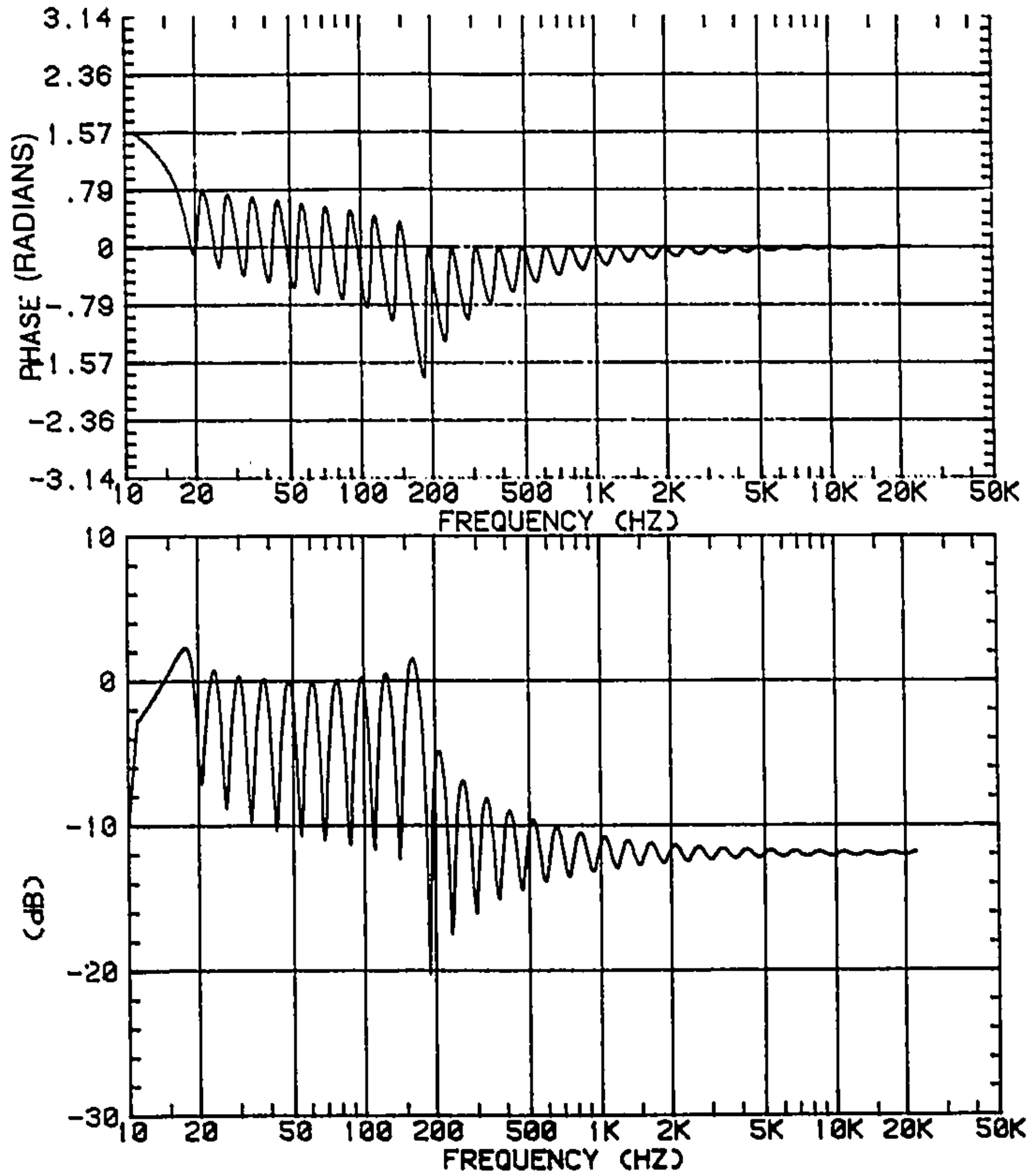


Figure 2.9 Real-Output-Weights Filterbank With High-Frequency Weights at -12 dB

Frequency (degrees)	mag of r	angle of r (degrees)
160.361769	0.103330	-2.593144
127.279221	0.082013	-2.347297
101.021584	0.065094	-2.223727
80.180885	0.051665	-2.143069
63.639610	0.041006	-2.078392
50.510792	0.032547	-2.019197
40.090442	0.025832	-1.959932
31.819805	0.020503	-1.896696
25.255396	0.016273	-1.826050
20.045221	0.012916	-1.744458
15.909903	0.010252	-1.647937
12.627698	0.008137	-1.531764
10.022611	0.006458	-1.390196
7.954951	0.005126	-1.216130
6.313849	0.004068	-1.000722
5.011305	0.003229	-0.732886
3.977476	0.002563	-0.398691
3.156925	0.002034	0.019425
2.505653	0.001615	0.543664
1.988738	0.001281	1.202197
1.578462	0.001017	2.030949
1.252826	0.000807	3.076080
0.994369	0.000641	4.397608
0.789231	0.000509	6.074980
0.626413	0.000404	8.216326
0.497184	0.000320	10.975128
0.394616	0.000254	14.583439
0.313207	0.000202	19.426693
0.248592	0.000160	26.243408
0.197308	0.000127	36.823530
0.156603	0.000101	58.242545

Table 2.2 Minimum Phase Filterbank Coefficients

2.2. Floating Point Simulation

2.2.1. Impulse Response

Figure 2.10 shows the allpass filterbank (coefficients as per Table 2.1) impulse response. It is seen that the effect of the non-linear phase response with flat magnitude setting is a smearing of the impulse input. Detailed examination of the impulse response reveals that the low frequencies are delayed with respect to the high frequencies as was indicated by its phase delay response shown in Figure 2.7.

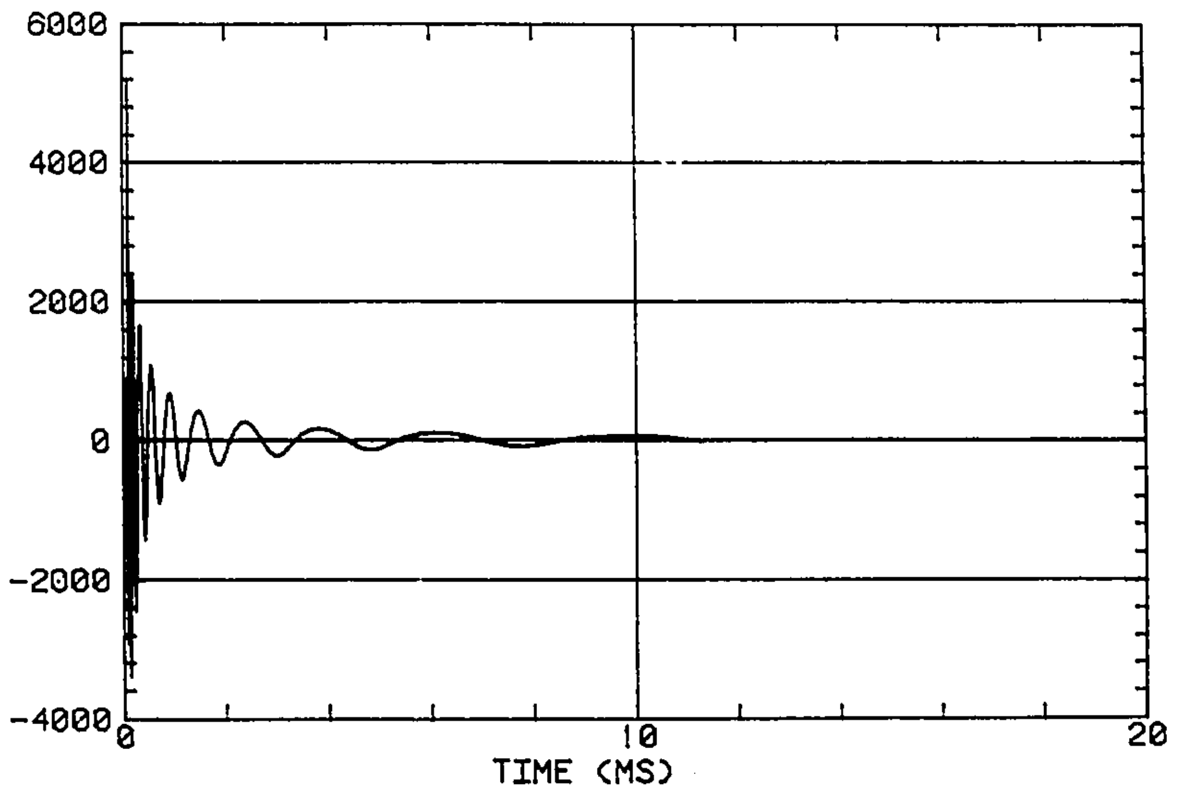


Figure 2.10 Allpass Filterbank Impulse Response

2.2.2. Subjective Listening Tests

The audibility of phase distortion has been a controversial topic in the field of high-fidelity audio. In the past, it was deemed that the human ear was

insensitive to phase distortion. Recently there is growing evidence that this is not the case. It is argued that indeed phase distortion, in the absence of any other forms of distortion, is audible and it may be in part due to the recent advent of very-high-fidelity recording and play-back equipment that other forms of distortion are not masking the effect of phase distortion. It is of course a much lesser factor than magnitude distortion.

Intuitively it is obvious that at some point, phase distortion must be audible. Take for example a music source that is separated into two bands : a high (say 1 kHz to 20 kHz) and a low one (say 20 Hz to 1 kHz). If the low band is delayed by one second and the high band is not, and this composite signal is played through a loudspeaker or a headset, clearly the result will not sound the same as the original source material ! Therefore phase audibility is a matter of degree. Now, whether or not the type of phase distortion encountered in most high-fidelity sound systems is above or below the human ear threshold is another question. However, what is important for our purposes is the audibility of the phase distortion produced by the allpass filterbank introduced in this chapter. In the case of a minimum-phase equalizer, the phase response of the input signal is not altered whenever the magnitude response is not either. Such is not the case for a non-minimum phase system.

Test signals were taken from various sources. Some signals like square and saw-tooth waves were generated using a synthesizer, while others were taken from three compact discs designed for audio testing. All signals were recorded on a digital audio tape (RDAT) in monophonic format; the digital audio tape stores audio using a 16-bit linear coding scheme. Then the digital information

on the tape was transferred to an IMB-AT compatible computer where the processing was done. The processing software was a floating point simulation of the filterbank. The fourteen tracks totalled about three and a half minutes using 18.8 Mbytes of storage. The floating point simulation required 11.1 minutes to process every second of audio. Please refer to Appendix B for the program listing.

Figures 2.11 shows the input and output waveforms for track 1. Notice the effect that phase distortion has on the input signal. The first cycle of the output waveform is not shown. The input waveform for track 2 is shown in Figure 2.12a. The output waveform for track 2 is shown in Figure 2.12b.

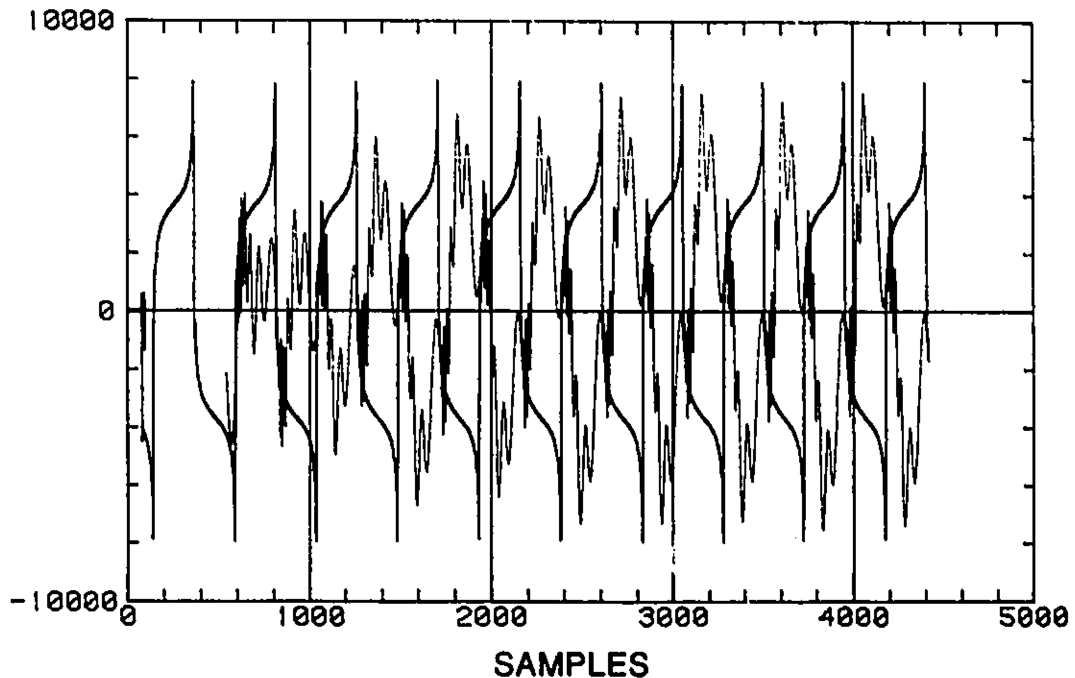


Figure 2.11 Allpass Filterbank Input and Output Waveforms for Track 1

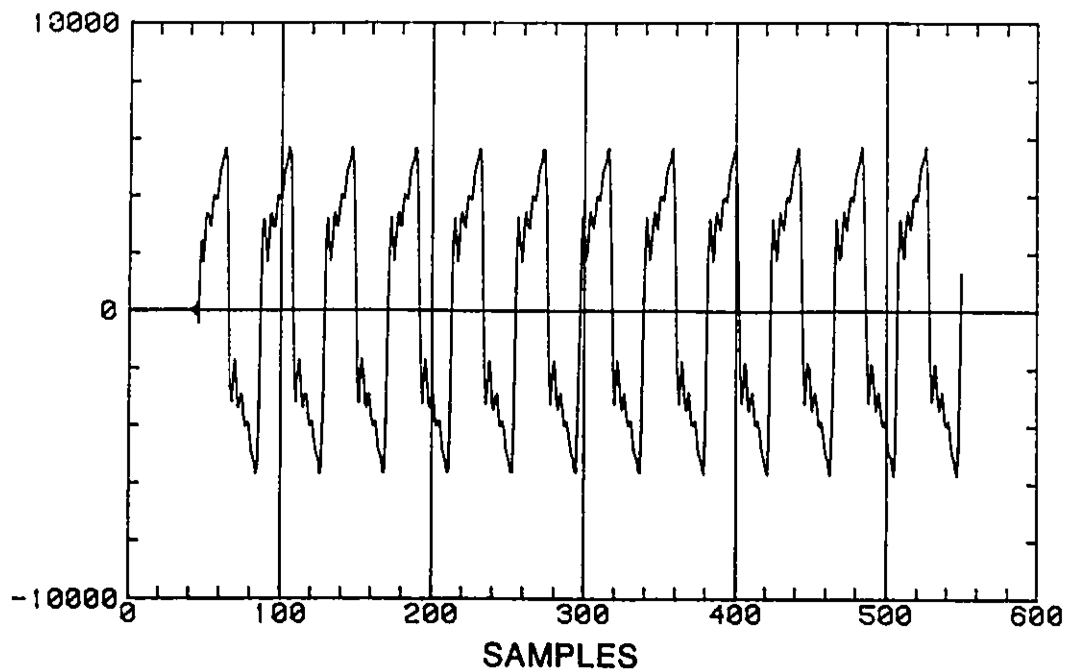


Figure 2.12a Allpass Filterbank Input Waveform for Track 2

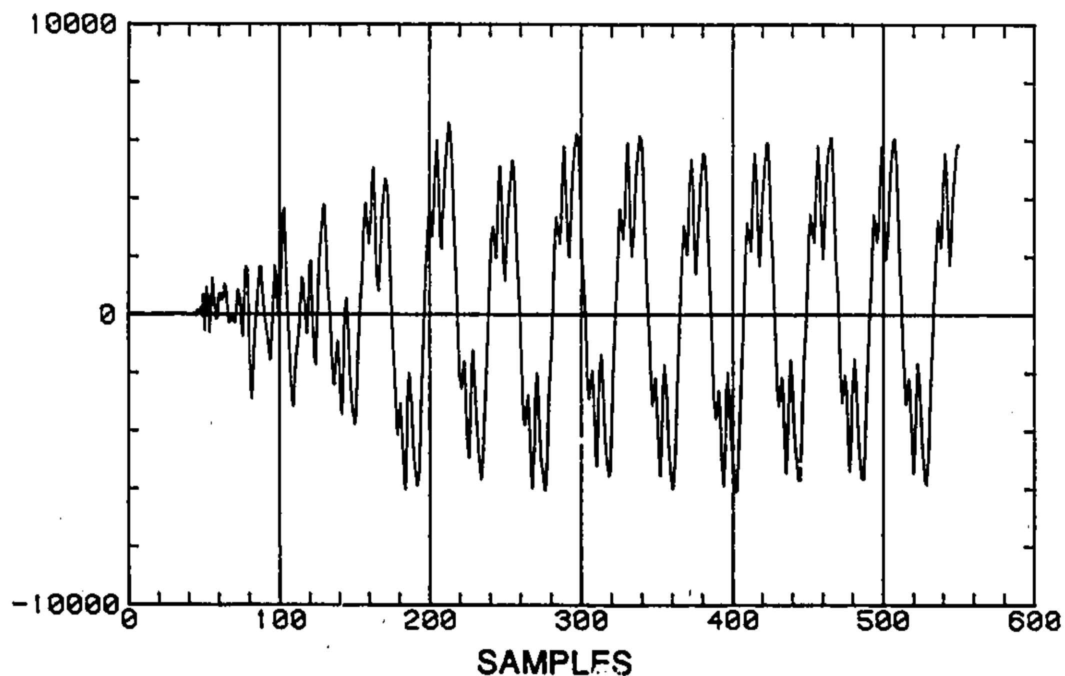


Figure 2.12b Allpass Filterbank Output Waveform for Track 2

Informal subjective tests were carried out with ten subjects. The sounds were played through headphones in mono and fed alternatively with the unprocessed and processed signals.

Fourteen test signals were used to determine if the allpass filterbank introduced any audible change in the sound. The selection of the types of signals used was influenced by what is deemed most noticeable in the literature [33] [34] [39] [40] namely square waves, clicks, dual sinusoids and unsymmetrical signals. Also some signals were included that are believed to be less susceptible to phase distortion: speech, choir and cymbals.

The test signals were :

Track 1 : synthesizer square wave with overshoot two G's below middle C (about 100 Hz)
Track 2 : synthesizer square wave with overshoot two octaves above middle C (about 1048 Hz)
Track 3 : synthesizer square wave with overshoot low E (about 41 Hz)
Track 4 : synthesizer square wave two G's below middle C
Track 5 : synthesizer unsymmetrical saw tooth two G's below middle C
Track 6 : synthesizer unsymmetrical saw tooth two octaves above middle C
Track 7 : synthesizer unsymmetrical saw tooth, fifth interval middle C and middle G; with Leslie effect rate : 5Hz, depth : 50 cents
Track 8 : EBU CD track 27 : castanets.
Track 9 : Yamaha test disc track 4 index 6 : kick drum.
Track 10: Yamaha test disc track 4 index 1 : tom.
Track 11: Yamaha test disc track 4 index 5 : crash cymbal.
Track 12: EBU CD track 50 : speech.
Track 13: EBU CD track 48 : choir.
Track 14: Digital Domain CD track 20 : pink noise.

EBU CD:	European Broadcasting Union Compact subjective assessment material. Disc no. 422 204-2.
Digital Domain :	The Digital Domain- A Demonstration, Elektra Records # 9 60303-2 1983.
Yamaha:	Yamaha DSP Test Disc #DSP 001.

There was very close agreement among all the listeners. Some test signals showed a significant difference between the original and the processed signals while others showed little or no difference. Tracks that showed a significant difference were tracks 1, 3, 5, and 9. Tracks that showed a slight difference were tracks 2, 4, and 6. Tracks that showed little or no difference were tracks 7, 8, 10, 11, 12, 13 and 14.

In all cases, the difference reported was an apparent increase in the low frequency content of the processed signal with respect to the original signal even though it was verified that the magnitude transfer function of the allpass system was exactly unity (by taking an FFT of the impulse response). Therefore the differences were attributed to the phase behavior of the allpass equalizer. In general the more low frequency content a signal had the more dramatic the increase in apparent low frequency content of the processed signal. This certainly makes sense since a signal devoid of low frequency content would be a poor test signal for evaluating the low frequency behavior of a system. It could be speculated that the fact that the low frequencies are delayed with respect to the high frequencies appears to have the effect of increasing the subjective low frequency content of some signals. However, this was not verified nor was the reason for such a phenomenon investigated.

It is interesting to note that Lipshitz et al. [33] found that low frequency square waves (150 Hz) displayed timbral changes as allpass networks were introduced. Hansen and Madsen [39] found a similar effect: phase distortion introduced a subjective timbral modification of low frequency (less than 1000 Hz) square waves.

The allpass filterbank was thus deemed a poor candidate for a graphic equalizer since in the absence of any magnitude equalization, the filterbank exhibited a subjective timbral modification for some types of signals.

Chapter 3

3. Parametric Equalizer

In light of the failure of the non-minimum phase filterbank as applied to graphic equalization, more conventional minimum-phase designs were investigated. In addition, it was thought that a parametric equalizer would be a better approach to audio equalization since fewer stages are usually required compared to graphic equalizers and thus more attractive from a processing power point of view. If on the other hand a graphic equalizer is required, it can be easily designed from a parametric equalizer.

The human ear responds approximately in a logarithmic (or equivalently in a dB) fashion to the amplitude of signals. Therefore, structures whose log-magnitude response is approximately linear in the gain term were investigated. In this way, at least one parameter of the equalizer could be adjusted directly in the dB domain.

Most of this study was carried out in the analog (Laplace) domain. The reason for this was to provide a logical progression from the work done by Bode on transmission line equalizers [41] [44] to the established analog equalization

techniques for audio. However, it was kept in mind that the results of this study would be ultimately implemented digitally. The digital implementation is dealt with at the end of this chapter.

3.1. Bode Equalizer

3.1.1. Background

In the 1930's many [41] [42] [43] studied the problem of equalization for transmission lines. Specifically, there was the need to equalize the attenuation of transmission lines whose amount expressed in decibels varied as a function of length, temperature and humidity. Since these could not be determined in advance, some form of automatic equalization was required.

Bode [41] [44] presented structures by means of which an arbitrary multiple of a given attenuation characteristic could be introduced into a circuit by changing the value of a single element. These equalizers had the property that their attenuation response with respect to a given attenuation curve was almost linear on a log-magnitude basis (or equivalently on a decibel basis) as a function of some adjustable impedance. This impedance was usually a variable resistor. The degree of this linearity was a function of the order of the circuit. The greater the order, the more linear the equalizer was on a decibel basis.

Ideally, these equalizers have the following property:

$$\ln T_i = \alpha_0(\omega) + f(x)\alpha_1(\omega)$$

where $\alpha_0(\omega)$ represents a reference response to which a fraction of $\alpha_1(\omega)$ is added or subtracted depending on the value of $f(x)$. In our case, the reference response is of no concern and we may simplify the above equation to :

$$\ln T_i = f(x)\alpha_1(\omega) \tag{3.1}$$

where $\ln T_i$ is in Nepers.

Bode has shown [41] [44] that a realizable network function approximating (3.1) is given by:

$$T_1 = \frac{[1 + xH_1(\omega)]}{[x + H_1(\omega)]} \tag{3.2}$$

where $0 \leq x \leq \infty$ and $H_1(\omega)$ is a normalized driving point impedance. Expanding (3.2) in terms of powers of $\ln H_1(\omega)$ results in [41]:

$$\ln T_1(\omega) = y \ln H_1(\omega) + C_3 \ln^3 H_1(\omega) + C_5 \ln^5 H_1(\omega) + C_7 \ln^7 H_1(\omega) + \ldots \tag{3.3}$$

where

$$C_3 = \frac{y}{12}\left(1 - y^2\right)$$
$$C_5 = \frac{y}{240}\left(1 - y^2\right)\left(2 - 3y^2\right)$$
$$C_7 = \frac{-y}{6720}\left(1 - y^2\right)\left(\frac{17}{3} - 20y^2 + 15y^4\right)$$

where $y = (x-1)/(x+1)$. (3.3) is a rapidly converging series and only the first two terms will be considered. The first term of (3.3) represents the ideal behavior of the equalizer as in (3.1) and the second term represents the first error term.

Brglez [45] applied a change of variables to (3.2) and suggested different network functions using:

$$T_2 = \frac{[1 - yH_2(\omega)]}{[1 + yH_2(\omega)]} \tag{3.4}$$

where:

$$x = \frac{[1 + y]}{[1 - y]}$$

$$H_1(\omega) = \frac{[1 - H_2(\omega)]}{[1 + H_2(\omega)]}$$

and where $-1 \leq y \leq 1$ and $0 \leq |H_2(\omega)| \leq 1$.

Therefore, (3.3) can be written as :

$$\ln T_1(\omega) = y \ln\left[\frac{[1 - H_2(\omega)]}{[1 + H_2(\omega)]}\right] + \frac{y}{12}\,(1 - y^2)\ln^3\left[\frac{[1 - H_2(\omega)]}{[1 + H_2(\omega)]}\right] + \ldots \tag{3.5}$$

expressed in loss in units of Nepers. (3.5) can be equivalently expressed in terms of gain as

$$\ln T_1(\omega) = y \ln\left[\frac{[1 + H_2(\omega)]}{[1 - H_2(\omega)]}\right] + \frac{y}{12}(1 - y^2)\ln^3\left[\frac{[1 + H_2(\omega)]}{[1 - H_2(\omega)]}\right] + \ldots \quad (3.6)$$

3.1.2. Biquadratic Equalizing Filters

An equalizing biquadratic filter has the following form:

$$H_{BE}(s) = \frac{s^2 + s\frac{A\omega_o}{Q} + \omega_o^2}{s^2 + s\frac{B\omega_o}{Q} + \omega_o^2} \quad (3.7)$$

B is usually equal to 1 such that A specifies the gain at the center frequency ω_o. Equation (3.7) is equivalent to (1.3) substituted into (1.1) and solving for a common denominator with $A = k + 1$ and $B = 1$. Equation (3.7) will be used to derive a form equivalent to Bode's equalizer. Q specifies the quality factor of the filter. However, since the boost at the center frequency of an equalizing biquad filter can be less than 3 dB, there is an ambiguity in interpreting Q. It can be interpreted as the Q of the equivalent bandpass filter with A and B set to 1. Nonetheless, to avoid any confusion, from here on, Q will simply refer to the parameter Q in equation (3.7) unless noted otherwise.

If $H(s)$ is evaluated at $s=j\omega$, the frequency response of the biquadratic filter is obtained:

$$H_{BE}(j\omega) = \frac{jQ[\omega^2 - \omega_o^2] + A\omega_o\omega}{jQ[\omega^2 - \omega_o^2] + B\omega_o\omega} \quad (3.8)$$

At this point it was hoped that the work by Bode and Brglez would provide a means of simplifying the automatic adjustment of the parametric equalizer by having an equalizing filter whose log-magnitude response would be almost linear in the gain term.

Both the numerator and denominator of (3.8) are divided by $\omega_o \omega$ to obtain:

$$H_{BE}(j\omega) = \frac{j\theta + A}{j\theta + B} \tag{3.9}$$

where $\theta = Q\left[\frac{\omega}{\omega_0} - \frac{\omega_o}{\omega}\right]$.

$A = 1 + y$ and $B = 1 - y$ are substituted and the top and bottom are divided by $1 + j\theta$ to obtain the following:

$$H(j\omega) = \frac{1 + \frac{y}{1 + j\theta}}{1 - \frac{y}{1 + j\theta}} \tag{3.10}$$

which is the same form as Brglez (3.4) except that it is expressed in terms of gain instead of loss. However, in order to be compatible with Bode and Brglez, y must have the range from -1 to 1. Therefore a gain variable G is introduced such that the gain at the center frequency is now $(1+yG)/(1-yG)$ and $H_2(\omega)$ in (3.4) becomes

$$H_2(\omega) = \frac{G}{1 + j\theta}$$

where $0 < G < 1$.

Thus, the Bode equalizer takes on the final form

$$H_{Bd}(j\omega) = \frac{1 + \dfrac{yG}{1 + j\theta}}{1 - \dfrac{yG}{1 + j\theta}} \tag{3.11}$$

Bode's approximation is applied as follows for second-order equalizing filters: the magnitude response in Nepers is the real part of (3.6)

$$\mathcal{Re}\left[ln\left(H_{Bd}(j\omega)\right)\right] \approx \mathcal{Re}\left[y\; ln\left(H_{ref}(j\omega)\right)\right]$$

or

$$ln\left|H_{Bd}(j\omega)\right| \approx y\; ln\left|H_{ref}(j\omega)\right| \tag{3.12}$$

where

$$H_{Bd}(j\omega) = \frac{j\theta + (1 + yG)}{j\theta + (1 - yG)} \tag{3.13}$$

$$H_{ref}(j\omega) = \frac{j\theta + (1 + G)}{j\theta + (1 - G)} \tag{3.14}$$

$H_{ref}(j\omega)$ is a reference curve which corresponds to the maximum boost desired. $H_{Bd}(j\omega)$ is a boost $(0 < y \leq 1)$ or cut $(-1 \leq y < 0)$ curve of lesser or equal

magnitude as $H_{ref}(j\omega)$ as a function of y. Therefore, the frequency response of a given curve is approximated by scaling another curve where y is the scaling factor. Typical frequency responses are shown in Figure 3.2 for the 12, 6, -6 and -12 dB cases. Pole-zero diagrams are shown in Figure 3.3.

3.1.2.1. Error Analysis

Rewriting (3.12) and including the next term as in (3.5)

$$ln|H_{Bd}(j\omega)| \approx y\, ln|H_{ref}(j\omega)| + \frac{y}{12}(1 - y^2)\, \mathcal{R}e\left[ln^3\left(H_{ref}(j\omega)\right)\right] \tag{3.15}$$

The first term of (3.15) is the approximation and the second term is taken to be the error. Since the error term is a cubic, the real part of it will get a contribution from both real and imaginary parts since:

$$\begin{aligned}\mathcal{R}e\left\{(d + fj)^3\right\} &= \mathcal{R}e\left\{d^3 + 3jd^2f - 3df^2 + jf^3\right\}\\ &= d^3 - 3df^2\end{aligned} \tag{3.16}$$

Therefore, the error term is:

$$error = \frac{y}{12}(1 - y^2)\, \mathcal{R}e\left[ln^3\left(H_{ref}(j\omega)\right)\right] = \frac{y}{12}(1 - y^2)(d^3 - 3df^2) \tag{3.17}$$

while the approximation is

$$approx = y\, ln|H_{ref}(j\omega)| = y\, d \tag{3.18}$$

$$\text{where } d = \ln\left[\frac{\sqrt{(1 - G^2 + \theta^2)^2 + 4\,\theta^2 G^2}}{(1 + G)^2 + \theta^2}\right] \tag{3.19}$$

$$\text{and} \quad f = \tan^{-1}\left[\frac{2\theta G}{1 - G^2 + \theta^2}\right]. \tag{3.20}$$

The error is 0 at $y = 0$, $y = 1$ and $y = -1$ which correspond to no gain, maximum gain and maximum loss respectively.

The error term is shown in Figure 3.1 for a 6 dB curve using a 12 dB reference curve ($y = 0.554$, $G = 0.6$) and a Q of 1. The maximum error at the center frequency is -0.740 dB.

The maximum error for any y actually occurs at $y = 0.577$ which corresponds to a boost of 6.28 dB at the center frequency for a reference boost of 12 dB ($G = 0.6$). This is found by taking the derivative of (3.17) with respect to y and setting the result to zero. For this value of boost, the error at the center frequency is -0.742 dB. This is very close to the value for the 6 dB case and therefore the error curve shown in Figure 3.1 can be taken to be the maximum error case for a 12 dB reference curve.

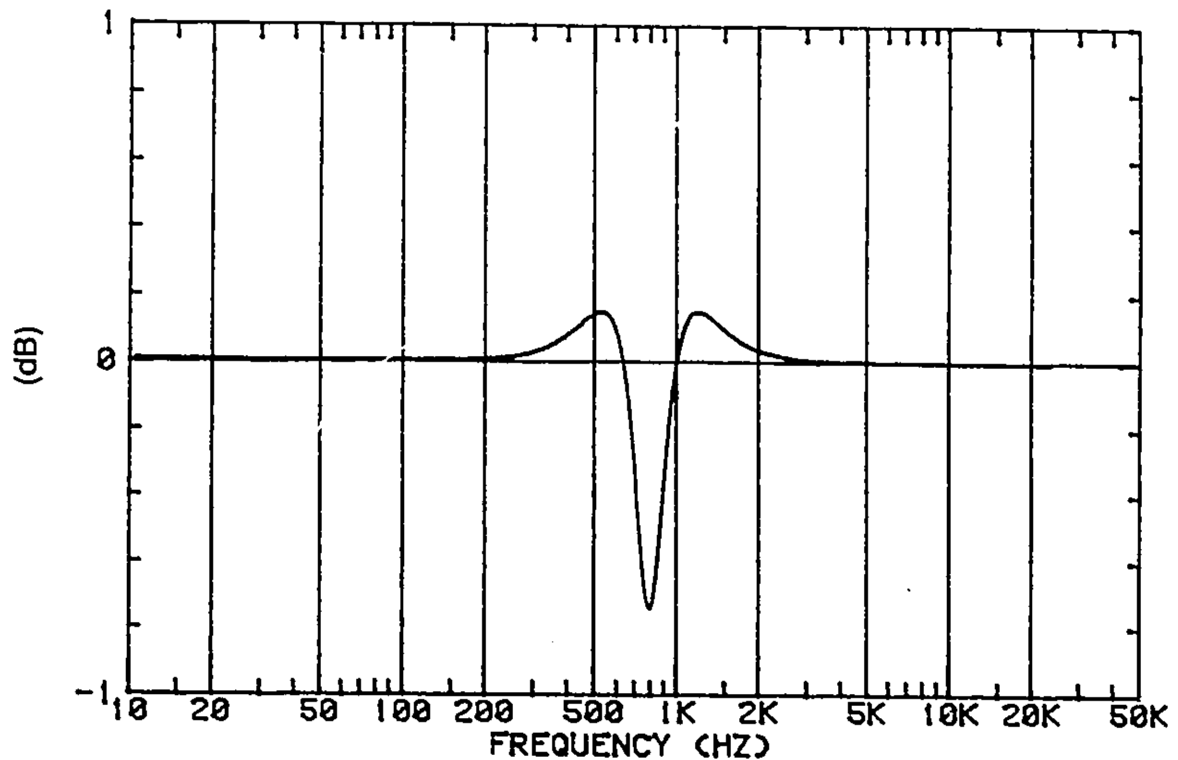


Figure 3.1 Bode Equalizer Error

Figure 3.1 shows the error function (3.17) obtained when a 12 dB boost curve is used as a reference to approximate a 6 dB curve for the case of a true Bode second-order equalizing filter.

3.1.2.2. Phase Response

It is clear that as long as A, B and Q are greater than zero, the response of (3.7) is minimum phase since the zeros lie in the left half of the s-plane.

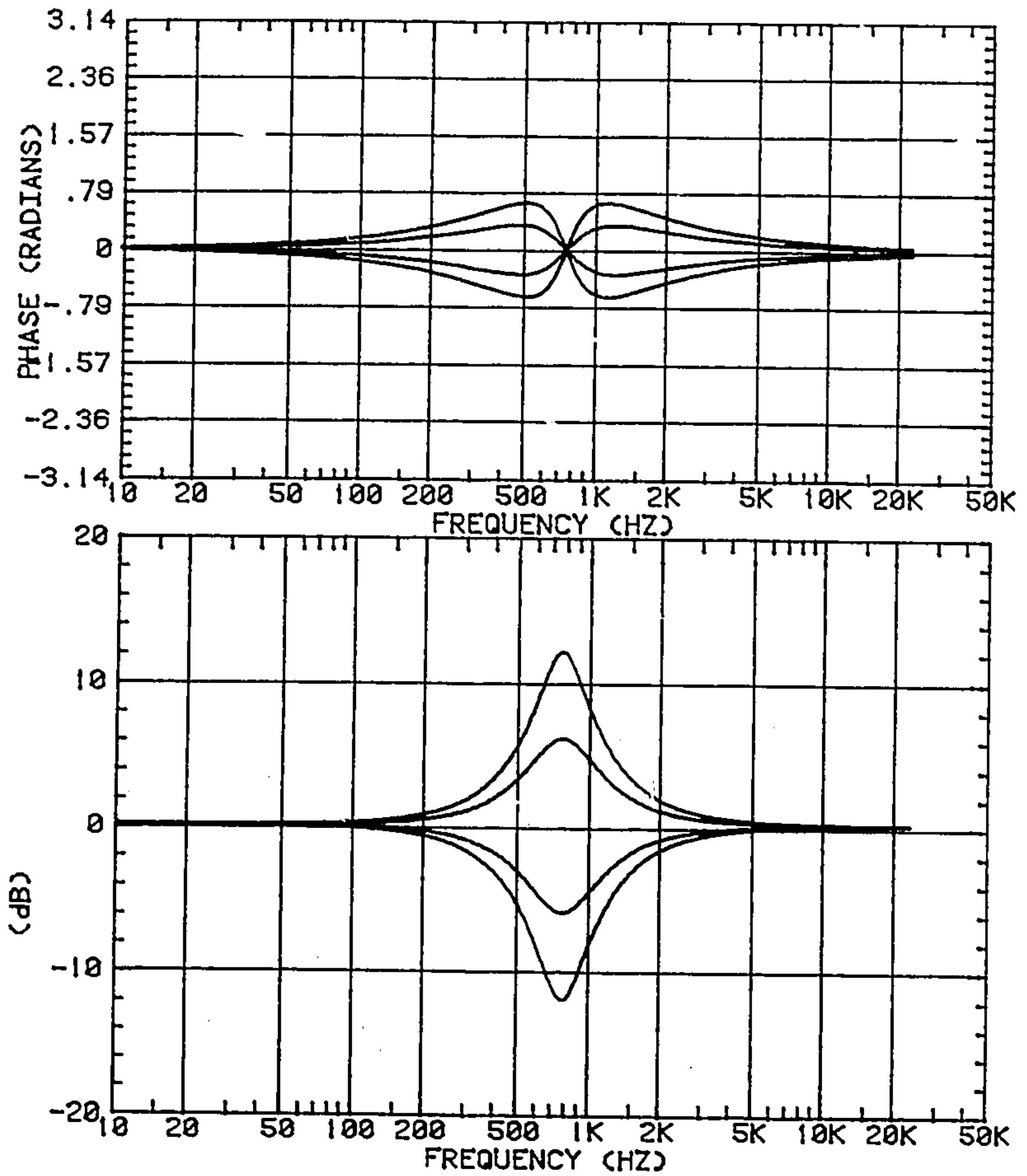


Figure 3.2 Bode Equalizer Typical Frequency Response Curves

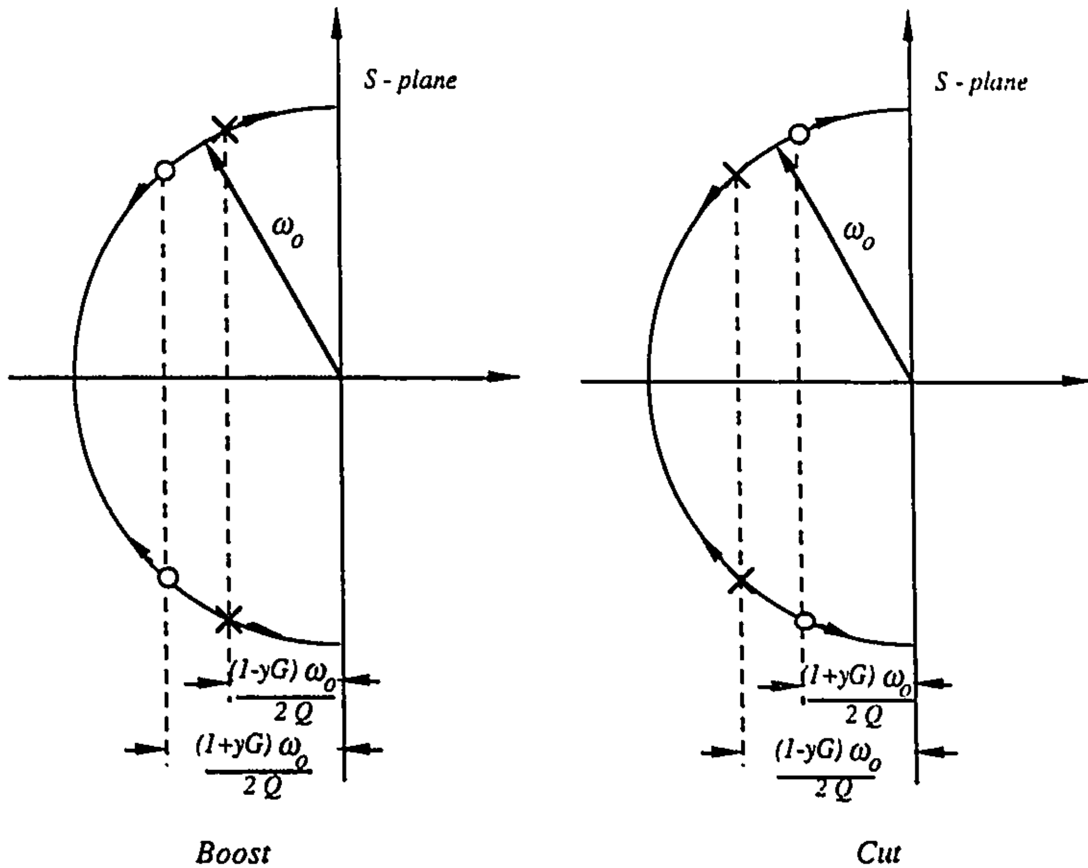


Figure 3.3 Bode Equalizer Pole-Zero Positions

3.1.2.3. Changing Q Value

It will be instructive to compare the Bode equalizer directly with a regular biquad filter. (3.7) can be rewritten (with $B = 1$) using Bode's substitutions to get the following

$$H_{bd}(s) = \frac{s^2 + s\,[1 + yG]\,\frac{\omega_o}{Q} + \omega_o^2}{s^2 + s\,[1 - yG]\,\frac{\omega_o}{Q} + \omega_o^2} \tag{3.21}$$

or equivalently

$$H_{bd}(s) = \frac{s^2 + s\left[\frac{1 + yG}{1 - yG}\right]\frac{\omega_o}{Q'} + \omega_o^2}{s^2 + s\frac{\omega_o}{Q'} + \omega_o^2} \qquad (3.22)$$

where

$$Q' = \frac{Q}{1 - yG} \qquad (3.23)$$

Equation (3.22) demonstrates that the second-order Bode equalizing filter can be treated as a regular biquad whose Q changes with gain.

3.1.2.4. Alternate-Error-Measure

In the early part of this study, an alternate-error-measure for the linearity of the gain term on a dB (or equivalently on a Neper) basis was used. That measure was to take the magnitude response in dB for some boost curve and compare this to the response for a reference curve divided by the ratio of the boosts at the center frequency. Of course, the Q's and center frequencies are the same. This alternate-error-measure was defined as:

$$error_{aem}(\omega) = \ln|H_{Bd}(\omega)| - m\ln|H_{ref}(\omega)| \qquad (3.24)$$

$$\text{where } m = \frac{\ln|H_{Bd}(\omega=\omega_0)|}{\ln|H_{ref}(\omega=\omega_0)|} \qquad (3.25)$$

In Bode's case

$$m = \frac{ln\left[\frac{1 + yG}{1 - yG}\right]}{ln\left[\frac{1 + G}{1 - G}\right]} \tag{3.26}$$

Equation (3.24) is the error incurred when the approximation

$$ln\left|H_{Bd}(j\omega)\right| = m\ ln\left|H_{ref}(j\omega)\right| \tag{3.27}$$

is used instead of (3.12).

This is basically the same as Bode's approach except that the scaling factor is equal to the ratio of the responses in Nepers (or dB) at the center frequency. This conveniently forces the error to be zero at the center frequency whereas it is maximum in the case of a true Bode equalizer. However, it is clear that this form is not as useful for analog implementation since the variable gain element of the equalizer would need to behave as (3.26).

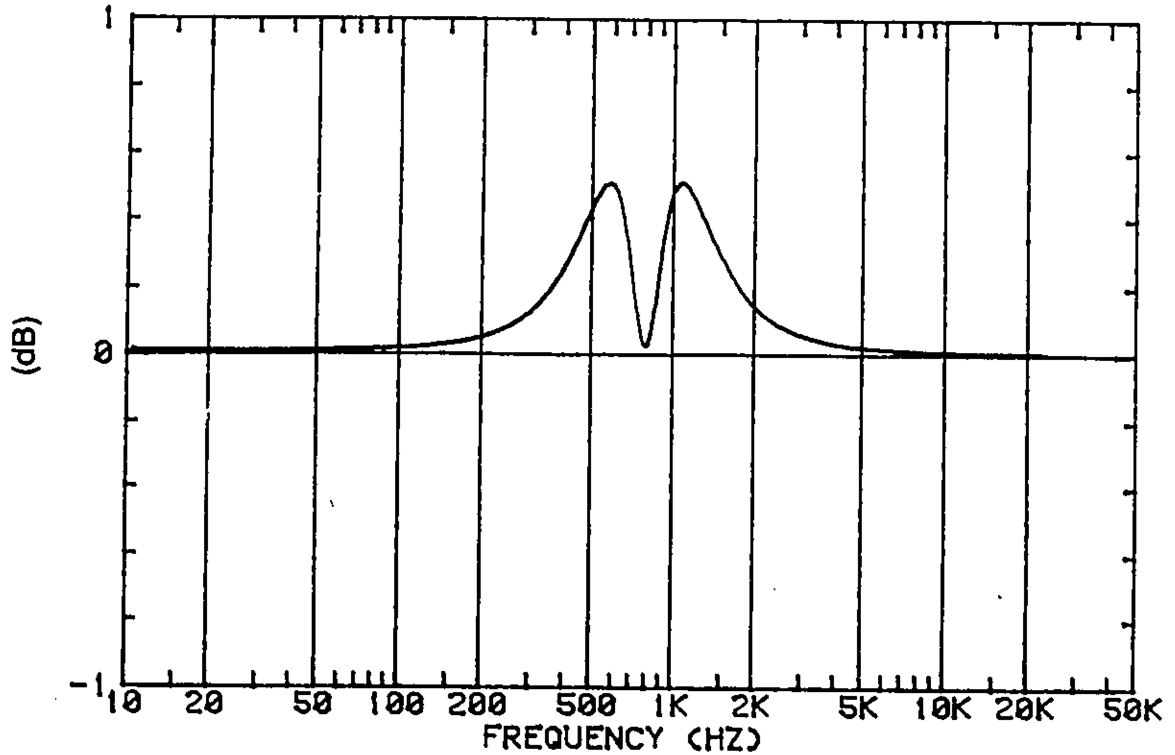


Figure 3.4 Alternate-Error-Measure: Bode

Figure 3.4 shows the alternate-error-measure as defined in (3.24) for the case of a 6 dB curve obtained from scaling a 12 dB curve where $m = 0.5$.

The advantage of the alternate-error-measure is that it allows direct comparison to a regular biquadratic filter (and another form as will be seen later).

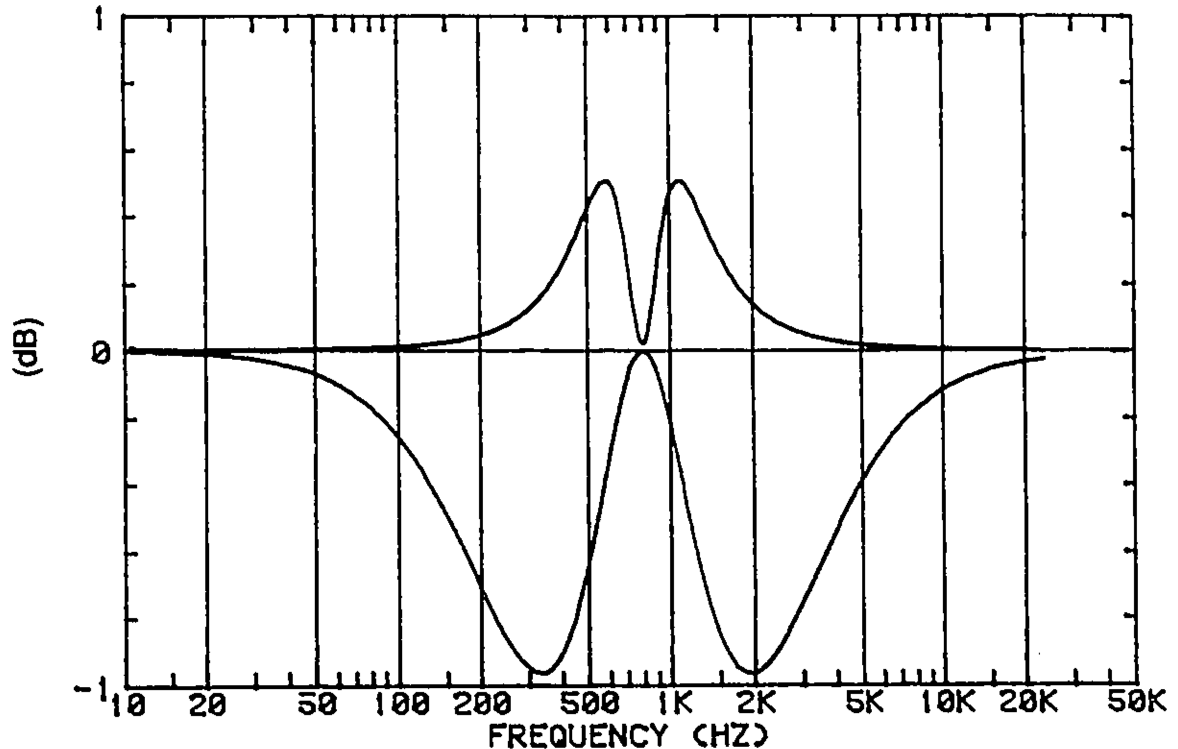


Figure 3.5 Alternate-Error-Measure: Bode and Regular Biquad

Figure 3.5 shows the alternate-error-measure for a 6 dB curve and a 12 dB reference curve for a regular biquad together with the Bode. Note that the Bode equalizer displays better linearity. It is seen that the effective change in the Q parameter in the Bode case as discussed in section 3.1.2.3 results in improved linearity in the gain term on a log scale.

3.2. Modified-Bode Equalizer

In the regular Bode equalizer as applied to a second-order biquad, the poles and zeros move away from one another or towards one another in a $1 + y$ and $1 - y$ fashion as shown in Figure 3.3. As an alternative to $A = 1 + y$ and $B = 1 - y$ (in eq. 3.7) like in the Bode case, a choice of $A = K$ and $B = 1/K$ was investigated. This choice turned out to be a judicious one. Not only does this yield to a more

linear behavior of the frequency response on a dB basis with respect to the gain parameter K, but also the approximation takes on a totally different form. Instead of the frequency response of a given gain setting being an approximation of another frequency response at a different gain setting, this new choice provides a direct algebraic approximation to the log-magnitude response. A proof of this follows.

3.2.1. Biquad Approximation

Let's put the equation in the form of (3.10). $A = y + 1$ has been substituted for $A = K$ and $B = 1 - y$ has been substituted for $B = 1/K$. Therefore, $y = 1 - K$ in the numerator of (3.10) and $y = 1 - 1/K$ in the denominator which yields the following:

$$H_{mBd} = \frac{1 + \dfrac{K - 1}{1 + j\theta}}{1 + \dfrac{1/K - 1}{1 + j\theta}} \tag{3.28}$$

and $$H_{mBd} = e^{\alpha}\left[\frac{1 + ve^{\alpha}}{v + e^{\alpha}}\right] \tag{3.29}$$

where the following substitutions have been made :

$$K = e^{\alpha},\ v = \frac{H_1}{1 - H_1},\ H_1 = \frac{1}{1 + j\theta}$$

Expanding (3.29) in terms of powers of $ln\ K$ yields [41]:

$$\ln H_{mBd} = \alpha + u\alpha + \frac{u}{12}\left(1 - u^2\right)\alpha^3 + \ldots \tag{3.30}$$

where

$$u = \frac{\nu - 1}{\nu + 1}$$

Higher-order terms in (3.30) are neglected as before. The magnitude response is found by taking the real part of the right side of (3.30):

$$\ln |H_{mBd}| = \frac{2\alpha}{1 + \theta^2} + \frac{4\alpha^3\theta^2\left(1 - \theta^2\right)}{3\left(1 + \theta^2\right)^3} + \ldots \text{ Nepers} \tag{3.31}$$

The first term of (3.31) is the approximation and the second term is taken to be the error. Therefore, the approximation to the log-magnitude frequency response is not a scaled version of another as in Bode's case but it is a direct algebraic approximation. Specifically:

$$20 \log \sqrt{\frac{K^2 + \theta^2}{1/K^2 + \theta^2}} \approx \frac{17.37\alpha}{1 + \theta^2} \text{ dB} \tag{3.32}$$

where the conversion factor 1 Neper = 8.686 dB was used and $\alpha = \ln K$.

(3.32) is in a form that is well suited for automation since the gain term α is linear on a log scale. Typical frequency responses are shown in Figure 3.6 for the 12, 6, -6 and -12 dB cases. Pole-zero diagrams are shown in Figure 3.7.

3.2.1.1. Error Analysis

Taking the derivative of the error term in (3.31) and setting it to zero, five extrema are found:

$$\theta_{1\text{-}2} = \pm 1.932,\ \theta_{3\text{-}4} = \pm 0.518,\ \theta_5 = 0 \text{ (center frequency).}$$

It should also be noted that the error is exactly zero at $\theta = 0$ and $\theta = \pm\ 1$. An error curve is shown in Figure 3.8 for the cases of $K = 1.41$ (6 dB) and $K = 2$ (12 dB) with $Q = 1$ and $f_o = 750$ Hz. The larger error curve corresponds to the 12 dB case. Each error curve is equiripple except at $\theta = 0$ where it is zero. The maximum error value is independent of center frequency and Q. It is only a function of K (or equivalently α). The position of the maximum error is of course a function of the center frequency and the shape of the error is a function of the Q. The magnitude of the maximum error, is found by substituting any of $\theta_{1\text{-}4}$ in (3.31):

$$max.\ error = 0.128\ \alpha^3\ Nepers\ =\ 1.11\ \alpha^3\ dB \tag{3.33}$$

The maximum error magnitude is strictly a function of the gain term. The higher the gain, the higher the error. It will be convenient to define an error in terms of a fraction of the maximum absolute boost or cut value (@ θ = 0):

$$e_{frac} = \frac{0.128\ \alpha^3}{2\alpha} = 0.064\alpha^2 \tag{3.34}$$

Equation (3.34) gives the maximum fractional error in terms of the peak boost or cut value when using the approximation (3.32). The maximum fractional error for the 12 dB case is $0.064\ (0.693)^2 \approx 0.031$. In dB this corresponds to 0.031×12 dB = 0.37 dB as shown in Figure 3.8. In the case of a 6 dB boost, the maximum fractional error is $0.064 \times (0.345)^2 \approx 0.0076$ which corresponds to about 0.05 dB.

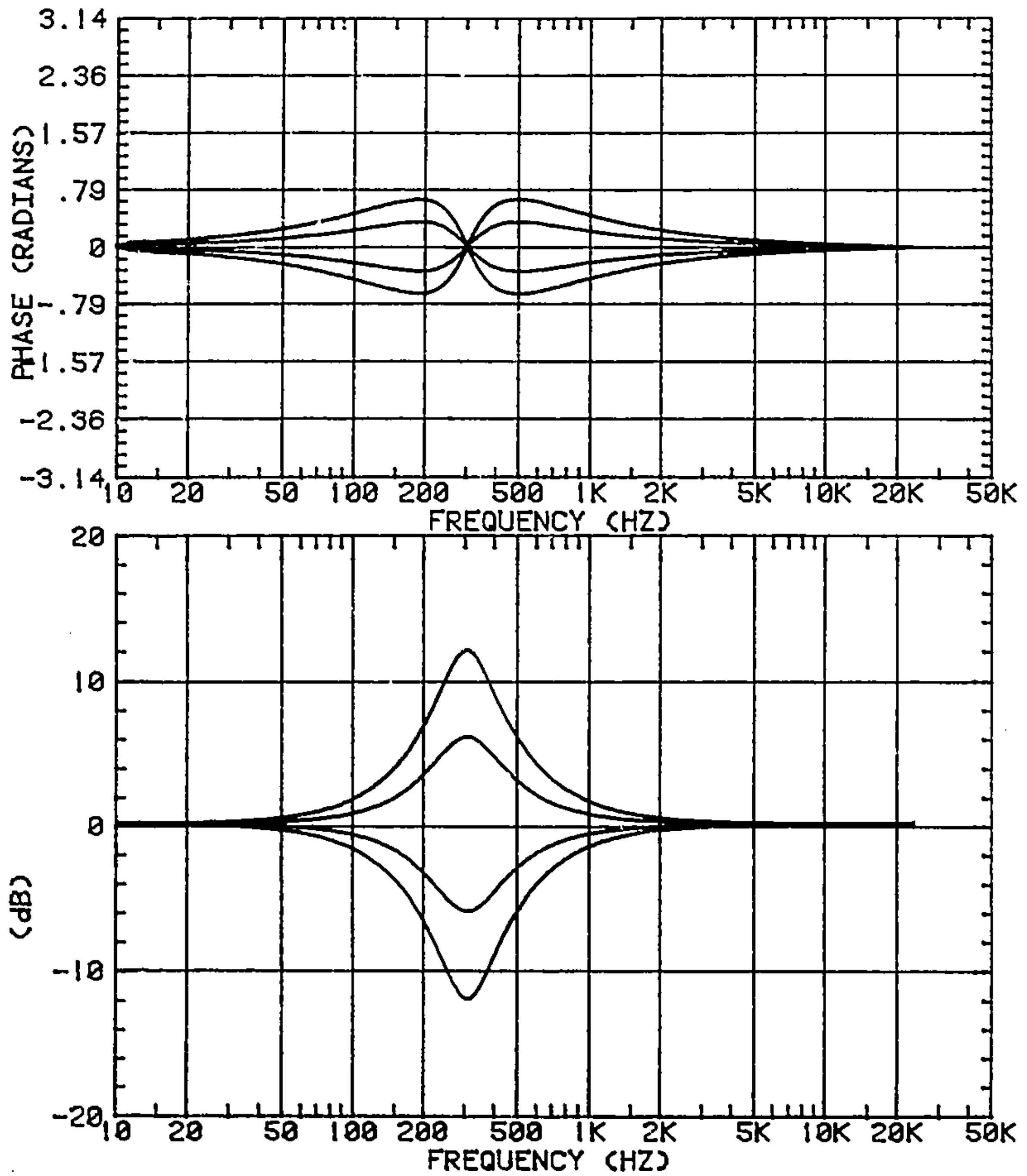


Figure 3.6 Modified-Bode Equalizer Typical Frequency Response Curves

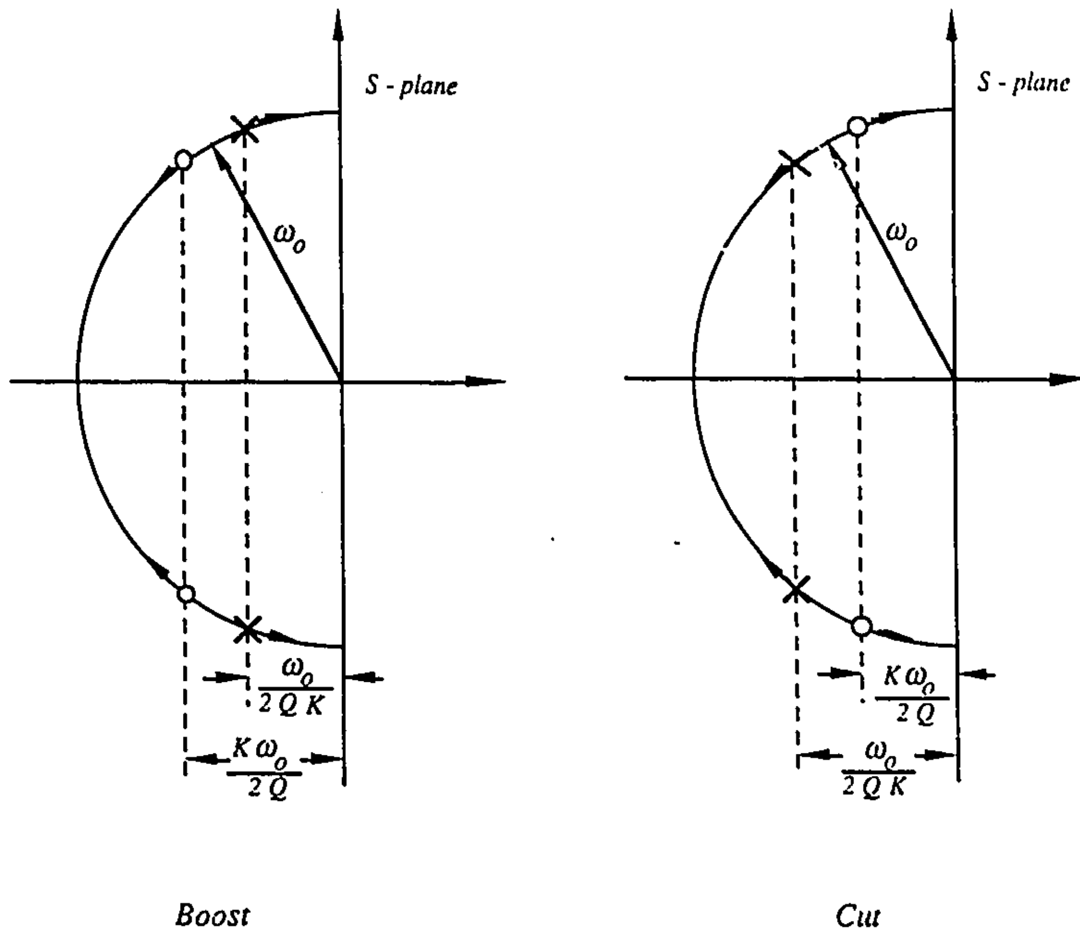


Figure 3.7 Modified-Bode Biquad Equalizer Pole-Zero Locations

3.2.1.2. Alternate-Error-Measure

It is instructive to compare the performance of a regular biquad, a Bode biquad and modified-Bode biquad using the alternate-error-measure (3.24). The error for each case is shown in Figure 3.9 for a 6 dB curve scaled from a 12 dB curve.

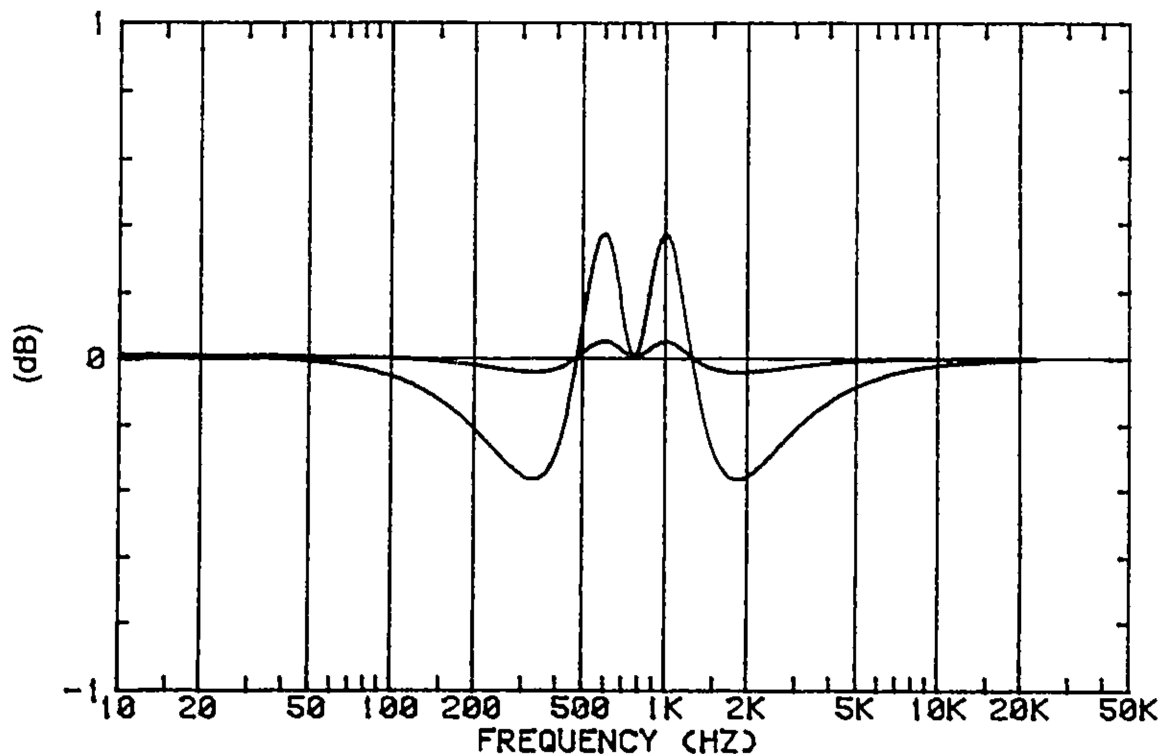


Figure 3.8 Modified-Bode Equalizer Error

Figure 3.8 shows the error function for a modified-Bode equalizing filter. The error function is defined by the second term of the series expansion (3.31) when approximation (3.32) is used. The larger curve corresponds to the 12 dB case (where the maximum error is 0.37 dB) while the smaller curve corresponds to the 6 dB case (where the maximum error is 0.05 dB).

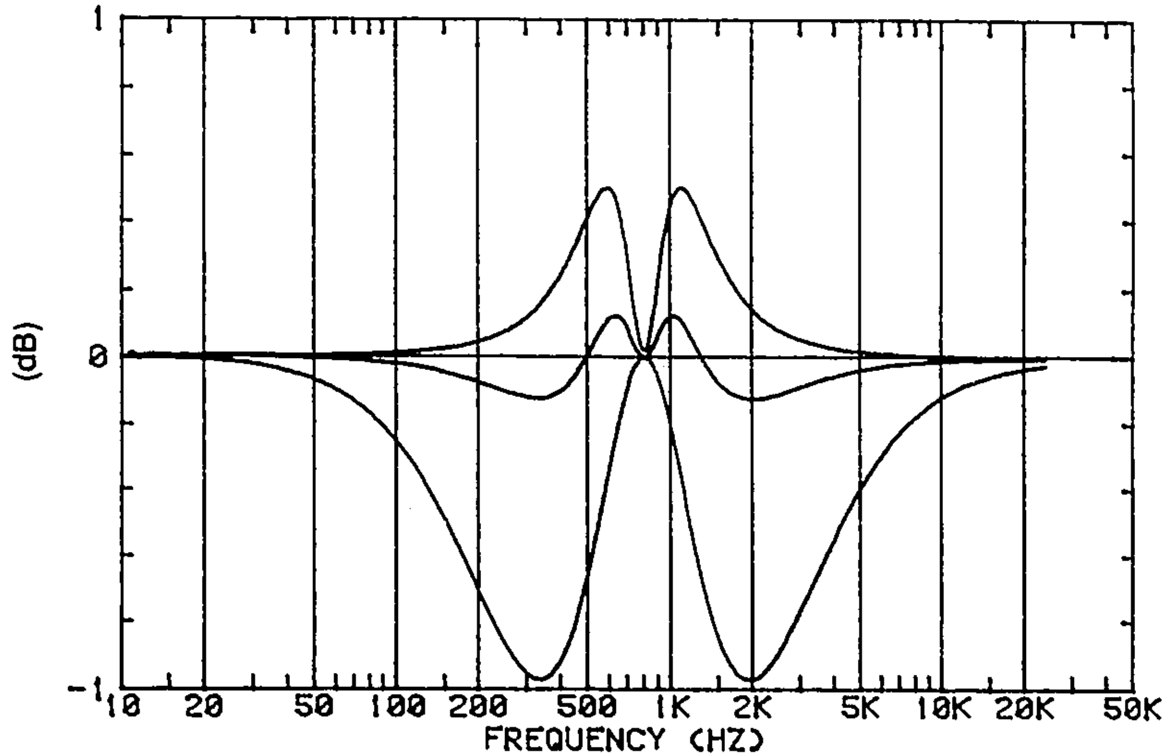


Figure 3.9 Alternate-Error-Measure: Bode, Biquad and Modified-Bode

Figure 3.9 shows the alternate-error-measure as defined in (3.24) for the regular biquad, Bode biquad and modified-Bode biquad when a 12 dB reference curve is used to approximate a 6 dB curve. The approximation to the 6 dB curve in each case is obtained by taking the 12 dB curve and dividing it by two. This approximation is then compared to the actual response and this difference is shown in Figure 3.9. The largest curve corresponds to the regular biquad and the smallest curve corresponds to the modified-Bode case.

3.2.1.3. Optimum Scaling Factor

The modified-Bode equalizer using K and $1/K$ as the scaling factors for the poles and zeros respectively give better linearity on a dB scale compared to the regular Bode using $1 + yG$ and $1 - yG$. The question arises then if a better

scaling factor exists. In an attempt to answer this question, an empirical study was carried out. A general error function of a form similar to (3.18) was defined:

$$error_{gef}(A,a,b,\theta) = 20\log\left|\frac{1+\frac{a}{1+j\theta}}{1+\frac{b}{1+j\theta}}\right| - x\,20\log\left|\frac{1+\frac{A}{1+j\theta}}{1-\frac{(3-A)/4}{1+j\theta}}\right| \qquad (3.35)$$

Then an optimum error function was defined as

$$error_{opt}(A,a) = min_{(b)}\left\{max_{(\theta)}\left|error_{gef}(A,a,b,\theta)\right|\right\} \qquad (3.36)$$

Equation (3.35) allows a boost curve with arbitrary scaling factors to be compared with a scaled arbitrary 12 dB curve. The second term of (3.35) with $x = 1$ corresponds to a 12 dB boost curve for any value of A. A allows the contribution of the real part of the numerator and denominator to be changed. $A = 0.6$ corresponds to a Bode equalizer while $A = 1$ corresponds to a modified-Bode equalizer. Parameter x is the ratio of the boost in dB at the center frequency.

(3.36) minimizes for any b the maximum peak error for any θ as a function of A and a. (3.36) was solved empirically and the program is listed in Appendix C. Figure 3.10 shows the values of b as a function of a satisfying (3.36). Comparing the first term of (3.35) with (3.28), it is found that the modified-Bode equalizer corresponds to

$$b = \frac{-a}{a + 1} \tag{3.37}$$

(3.37) is also shown in Figure 3.10. The curve coincides with the solution to (3.36) for $A = 1$. Therefore, in this sense, the choice of K and $1/K$ as the scaling factors for the poles and zeros as in (3.28) is optimum. Figure 3.10 also shows the optimum b versus a values for $A = 0.8$ and $A = 0.6$. The latter corresponds to the Bode case. Comparing the first term of (3.35) with (3.10), it is found that the Bode case corresponds to

$$b = -a \tag{3.38}$$

Clearly the Bode equalizer is not optimum in the sense of (3.36) when applied to biquadratic filters.

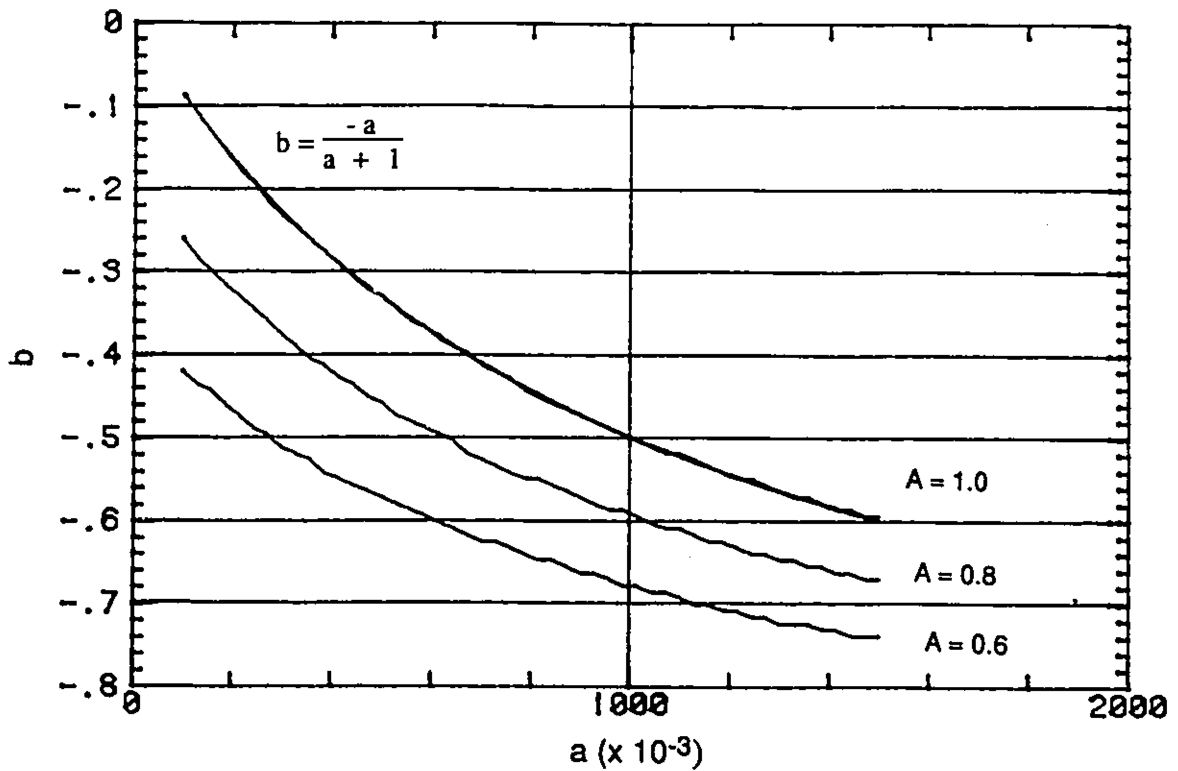


Figure 3.10 Optimum Error Parameter Values

3.2.2. First-Order Shelving Approximation

Some parametric equalizers include one or more shelving filter stages. Those are first-order filters of the form

$$H_s(s) = \frac{s + \omega_z}{s + \omega_p} \tag{3.39}$$

Equation (3.39) can be converted to a modified-Bode form as follows

$$H_s(s) = \frac{s + K\omega_o}{s + \omega_o/K} \tag{3.40}$$

A pole-zero plot of (3.40) is shown in Figure 3.11. The pole and zero both sit on the real axis on the left-hand side of the s-plane. For a boost ($K > 1$), the pole is closer to the $j\omega$ axis than the zero and they move in opposite direction. The reverse occurs for a cut ($K < 1$): the zero is closer to the $j\omega$ axis than the pole.

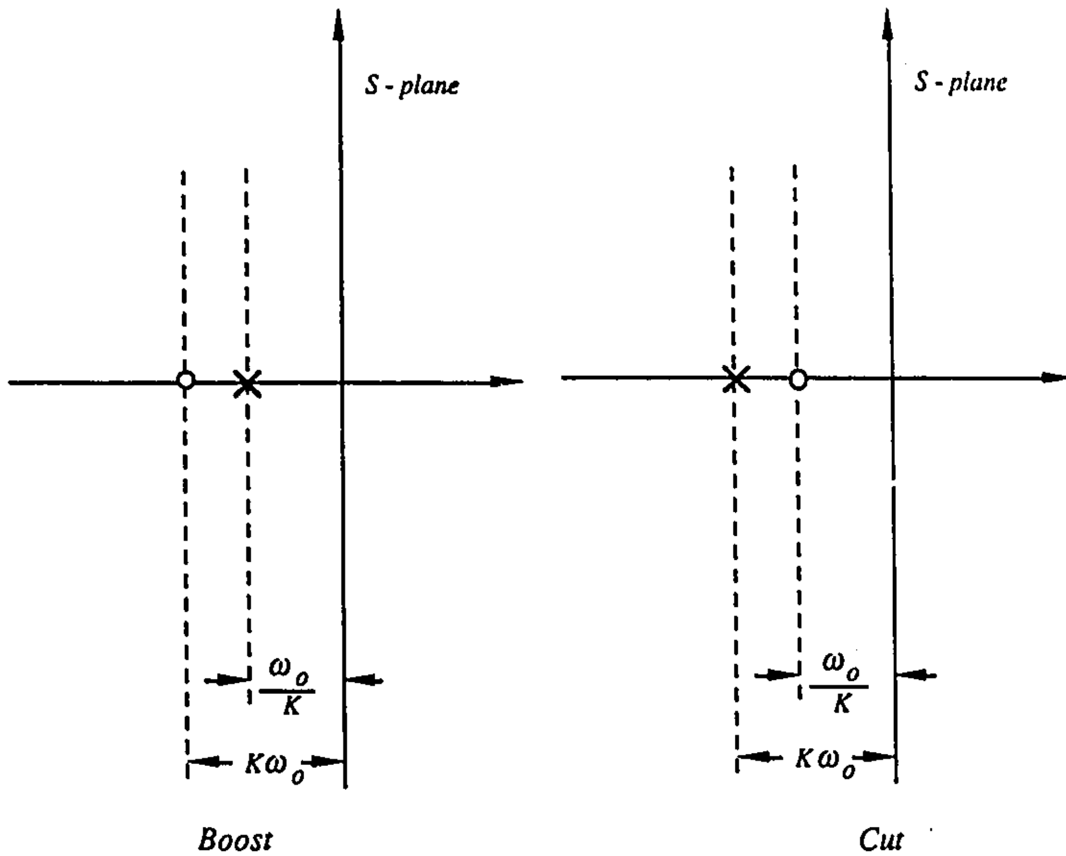


Figure 3.11 Modified-Bode First-Order Equalizer Pole-Zero Locations

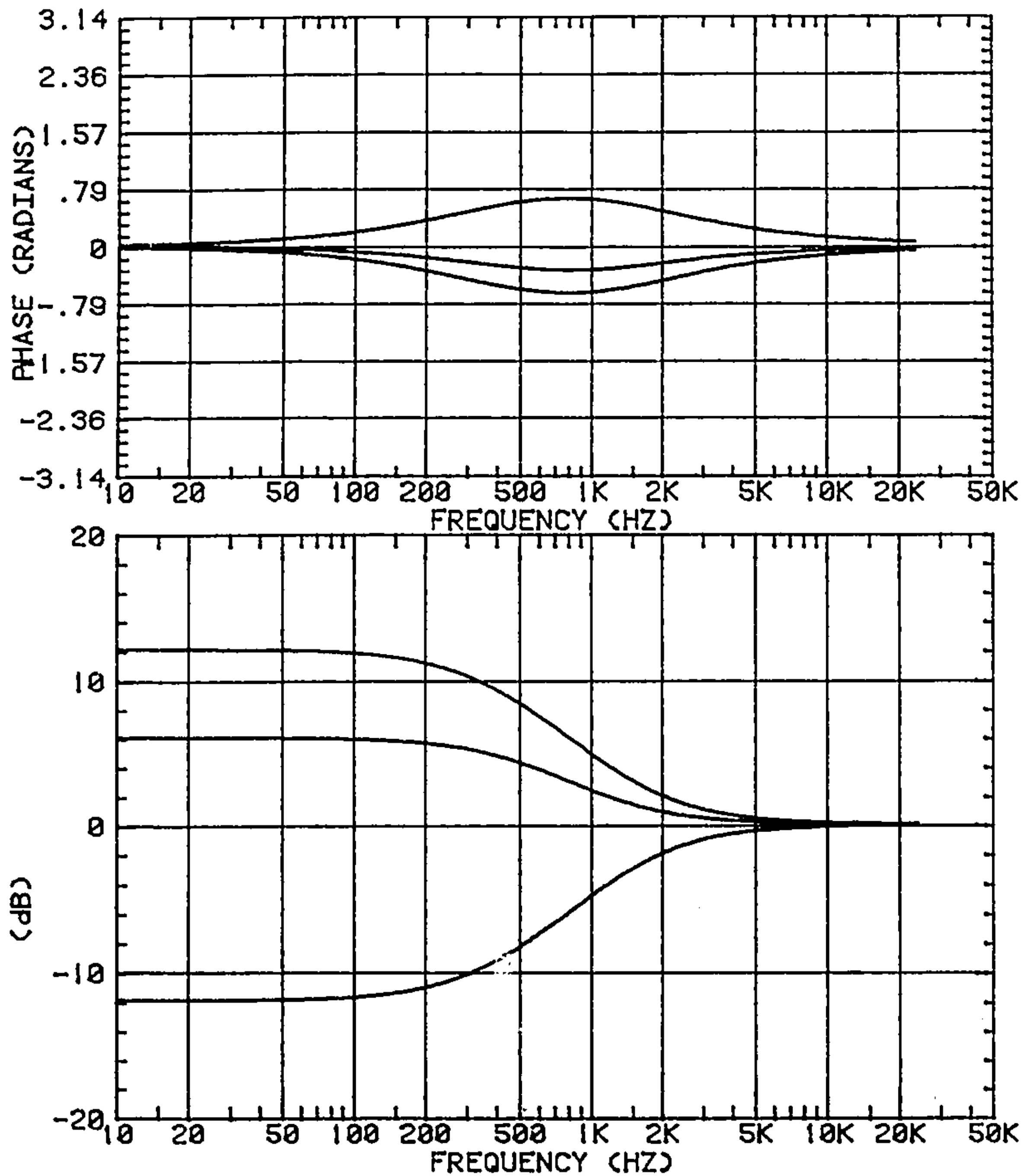


Figure 3.12 First-Order Shelving Frequency Response

The frequency response is found by substituting $s=j\omega$ in (3.40) with the following result

$$H_s(j\omega) = \frac{j\dfrac{\omega}{\omega_o} + K}{j\dfrac{\omega}{\omega_o} + 1/K} \tag{3.41}$$

Typical frequency response curves are shown in Figure 3.12 for the 12, 6 and -12 dB cases with f_o = 750 Hz.

3.2.2.1. Error Analysis

The approximation and the error take on exactly the same form as (3.31) except that

$$\theta = \frac{\omega}{\omega_o} \text{ instead of } Q\left[\frac{\omega}{\omega_o} - \frac{\omega_o}{\omega}\right] \tag{3.42}$$

Error curves are shown in Figure 3.13 for the 12 and 6 dB cases with f_o = 750 Hz. The maximum error is the same as in the second-order case but the shape is different. It has only two peaks.

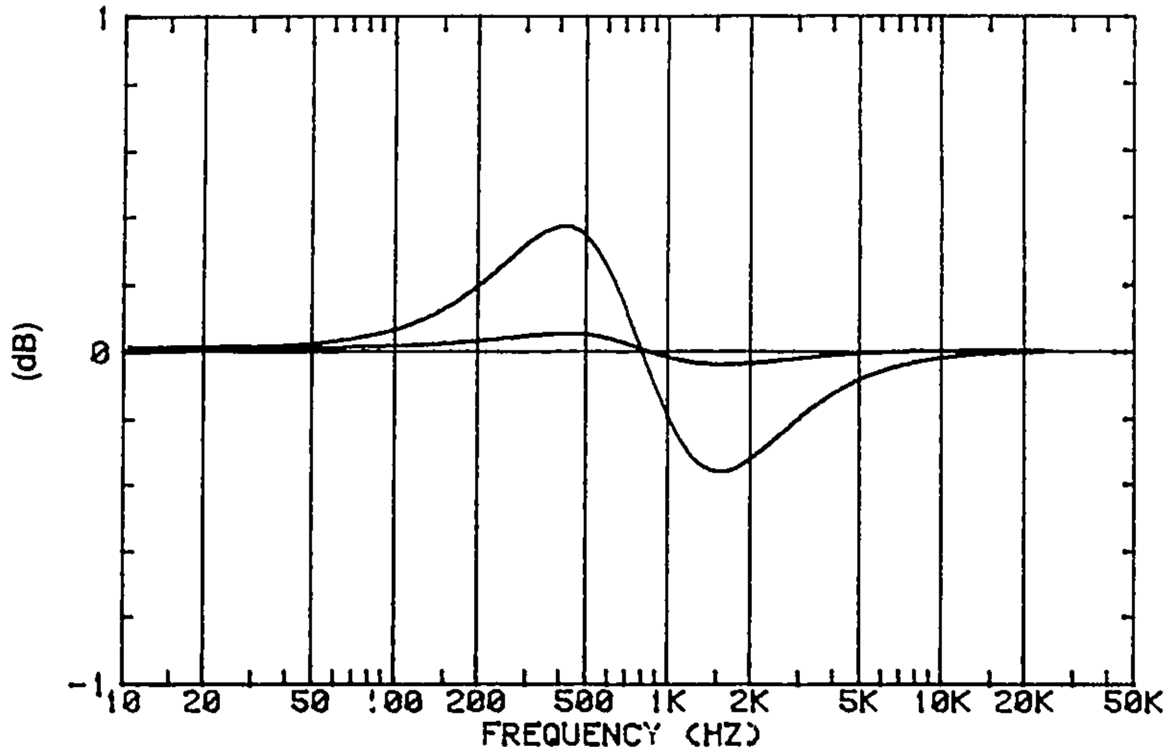


Figure 3.13 First-Order Shelving Error Curve

3.2.3. Automation Method 1 - Biquads

A procedure was required whereby the gain, center frequency and *Q* parameters of a biquad equalizing filter could be optimized using frequency response data. Each biquad would be optimized in succession such that a cascade of these with the original frequency response measurement would match some target response.

Bullis and Brubaker [46] have proposed a linearized least-squares estimation technique which had the same goals as stipulated above. The filters used were analog first and second-order equalizing filters. Their "two-parameter" estimation technique was adapted to the modified-Bode second-order equalizing filter. A summary of this technique follows.

The measurement in dB (m_n) at the nth frequency is set equal to the modified-Bode approximation function (3.32):

$$m_n = \frac{17.34\alpha}{1 + Q^2\left[\frac{f_n}{f_o} - \frac{f_o}{f_n}\right]^2} \tag{3.43}$$

Multiplying both sides of (3.43) by $1 + jQ^2\left[\frac{f_n}{f_o} - \frac{f_o}{f_n}\right]^2$ gives

$$m_n = 17.34\alpha - m_nQ^2\left[\frac{f_n}{f_o} - \frac{f_o}{f_n}\right]^2 \tag{3.44}$$

The following vectors are defined:

$$g = \begin{bmatrix} m_0 \\ m_1 \\ \cdot \\ \cdot \\ \cdot \\ m_{N-1} \end{bmatrix} \tag{3.45}$$

$$a = \begin{bmatrix} 17.34 \\ 17.34 \\ \cdot \\ \cdot \\ \cdot \\ 17.34 \end{bmatrix} \tag{3.46}$$

$$b = \begin{bmatrix} -m_0\left[\frac{f_0}{f_o} - \frac{f_o}{f_0}\right]^2 \\ -m_1\left[\frac{f_1}{f_o} - \frac{f_o}{f_1}\right]^2 \\ \cdot \\ \cdot \\ \cdot \\ -m_{N-1}\left[\frac{f_{N-1}}{f_o} - \frac{f_o}{f_{N-1}}\right]^2 \end{bmatrix} \tag{3.47}$$

A system of equations in matrix form is formulated:

$$g = H\ x \tag{3.48}$$

where

$$H = [a\ b] \tag{3.49}$$

$$x = \begin{bmatrix} \alpha \\ Q^2 \end{bmatrix} \tag{3.50}$$

The best least-squares solution of (3.48) is

$$x = [H^t H]^{-1} H^t g \tag{3.51}$$

There are two disadvantages to this method. The first is that it does not always give realizable filters and the second and fatal disadvantage is that the solution is not really the best least-squares solution to the desired system. The cross-multiplication by $1 + jQ^2\left[\frac{f_n}{f_o} - \frac{f_o}{f_n}\right]^2$ changes the error formulation. The true sum of the squared errors (SSE) is

$$SSE = \sum_{n=0}^{N-1} \left(m_n - \frac{17.34\alpha}{1 + \theta_n^2}\right)^2 \tag{3.52}$$

But the SSE that is minimized using Bullis and Brubaker's method is

$$SSE = \sum_{n=0}^{N-1} \left(m_n[1 + \theta_n^2] - 17.34\alpha \right)^2 \tag{3.53}$$

Experiments showed that even with simple, single-bump frequency response curves, the method does not converge to the real least-squares solution. The method does converge to the true least-squares solution when the data is exact. That is, if a frequency response curve is generated using values for gain, center frequency and Q and this data is used to find the best gain and Q given the correct center frequency using (3.51), then the method yields the correct values for gain and Q. However, the method fails using even simple real frequency data curves. Therefore an alternate method was required.

3.2.4. Automation Method 2 - Biquads

The failure of the first method adapted from Bullis and Brubaker demonstrated the importance of the error formulation. It must represent exactly what needs to be minimized.

3.2.4.1. Gain Term

From (3.52) it is clear that a solution can be found for α by differentiating with respect to α and setting the result equal to zero:

$$\frac{\partial SSE}{\partial \alpha} = -2 \sum_{n=0}^{N-1} \left(m_n - \frac{17.34\alpha}{1 + \theta_n^2} \right) \frac{17.34}{1 + \theta_n^2} = 0 \tag{3.54}$$

And the solution is

$$\alpha = \frac{1}{17.34} \frac{\sum_{n=0}^{N-1} m_n}{\sum_{n=0}^{N-1} \left(1 + \theta_n^2 \right)} \tag{3.55}$$

where here again

$$\theta = Q\left[\frac{f_n}{f_o} - \frac{f_o}{f_n}\right] \text{ and } \alpha = \ln K.$$

(3.55) gives a direct solution for the gain term of a biquadratic filter given a center frequency and Q. A direct closed form solution for Q and f_o could not be found. However, two separate methods were devised to find suitable values for f_o and Q.

3.2.4.2. Center Frequencies

One method of finding center frequency candidates is directly from the frequency response data. It consists of taking the derivative of the frequency response in the dB domain. The frequencies where the derivative crosses zero correspond to a peak or dip in the frequency response. This method gave very good results. The derivative was approximated using a backward difference equation.

3.2.4.3. Q

An iterative method was used to find the best Q value. Values for Q's are scanned between a lower and upper limit and the one giving the smallest SSE is used. As an example, values of Q are scanned between 0.5 and 10 in steps of 1.0. Then a fine scan is performed around the best Q in steps of 0.1. In this fashion, the best Q value is found with a resolution of 0.1 in 30 iterations.

3.2.4.4. Frequency Response Weighting

In the absence of some form of frequency weighting, the best least-squares solution for a single stage to a given frequency response is often a flat gain curve going through the average measurement response (in the case of a flat target frequency response). This arises because only one stage is optimized at a time and often the single best stage with the best least-squares error is a flat gain stage. However, since it is known that more than one stage will eventually be used, the algorithm can be forced to concentrate on each individual bump and dip in the frequency response. One way to do this is to apply a weighting curve centered around each center frequency. The difference between the target in dB and the measurement in dB is taken and the result is multiplied by the weighting curve. A rectangular weighting curve was used in our case with height of 1 at the center frequency and extending on each side to the 0 dB crossings of the curve resulting from the difference between the target and the measurement. Outside this region the weighting curve was 0.

3.2.4.5. General Description

Figures 3.14a-c describe in a block diagram form, how the best gain and Q parameters are found for a biquad stage given a center frequency f_o. First, the target equalization curve is weighted around the given f_o. The target equalization curve is the difference in dB between the target frequency response and the unequalized system frequency response. Then Q is assigned a lower limit value (in our case 0.5). The best least-squares gain term α (and K) is found using (3.55). Then the SSE between this stage and the target equalization curve is found. This procedure is repeated for every Q value between an upper and lower limit (in our case 0.5 and 10 were used respectively) in steps of 0.5 and the Q and K parameters resulting in the lowest SSE are kept.

The above procedure is repeated again using a smaller step size for Q using the best Q value found in the previous procedure as a starting point. A fine scan is performed on each side of this gross Q value and a fine Q value is thus obtained.

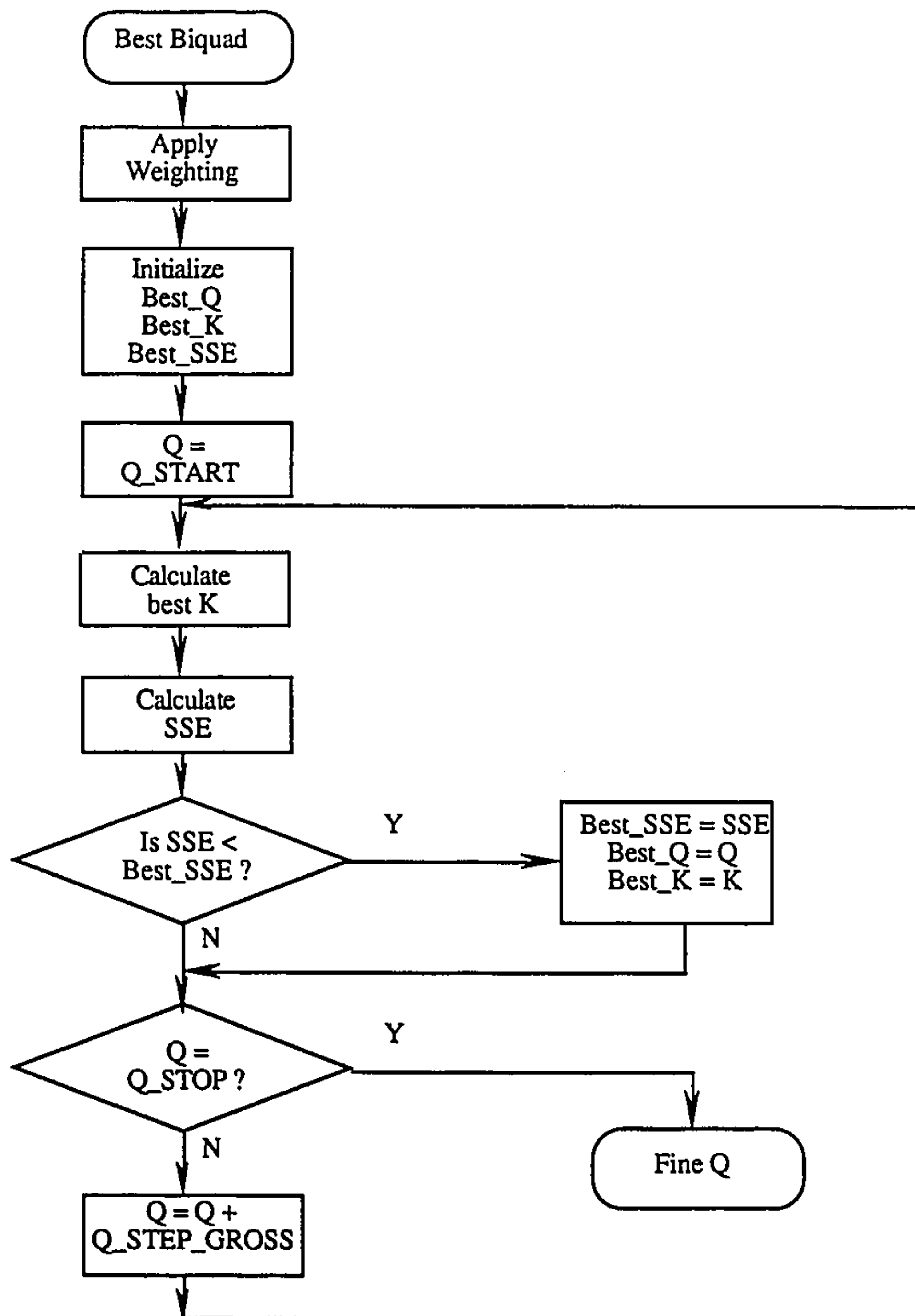


Figure 3.14a Best Biquad Stage Algorithm

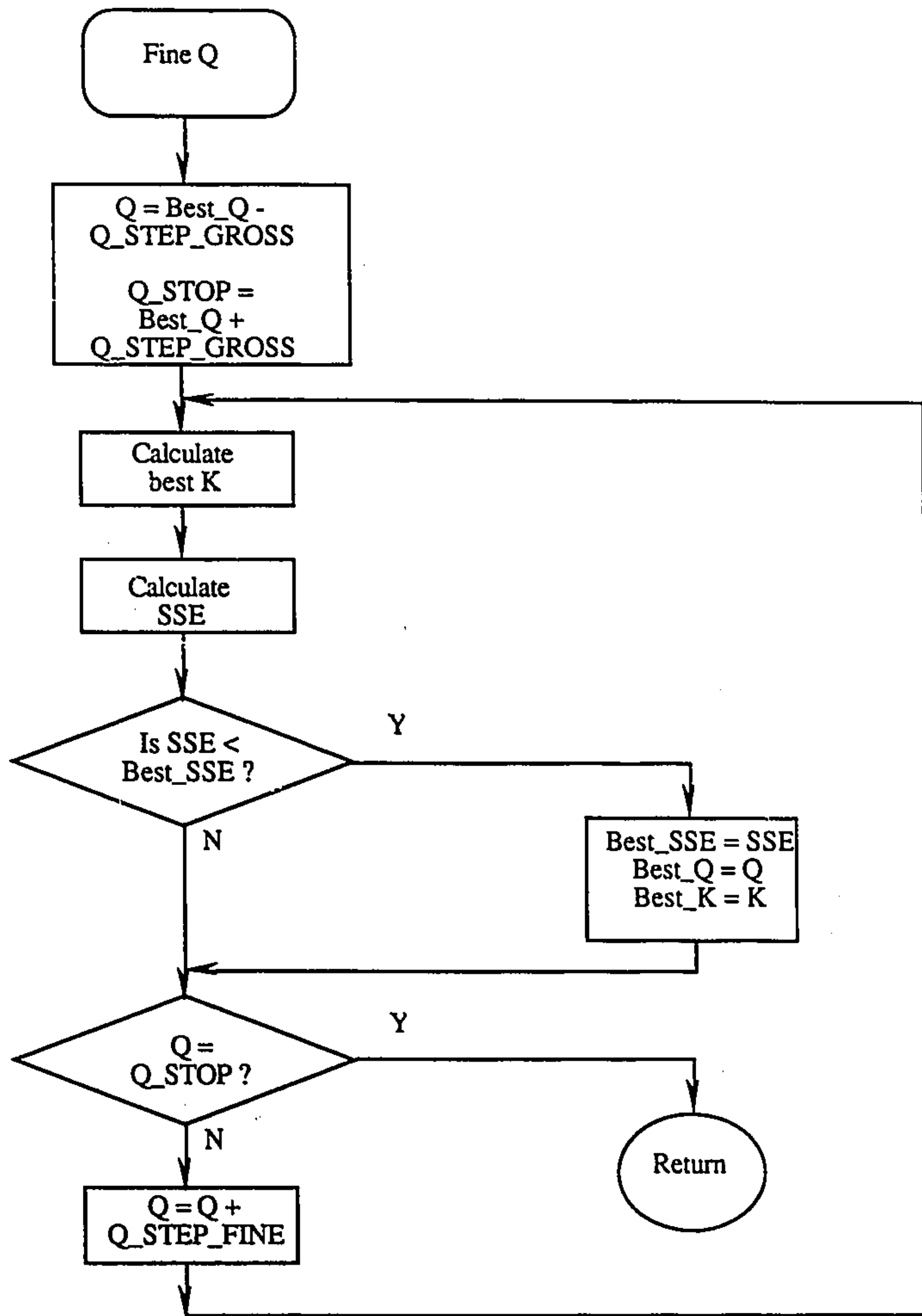


Figure 3.14b Best Biquad Stage Algorithm

3.2.5. Automation Method 2 - Shelving

3.2.5.1. Gain Term

The best gain term α for a shelving filter is found by using equation (3.55) and substituting

$$\theta = \frac{f_n}{f_o} \tag{3.56}$$

3.2.5.2. Center Frequencies

The best center frequency for a shelving filter can be found using an iterative method. The center frequency parameter is set equal to the first measurement frequency. The best least-squares gain term is found using (3.55) and (3.56) and the resulting SSE is calculated and stored. Then the center frequency parameter is set equal to the second measurement frequency and a new best least-squares gain term and the resulting SSE are calculated. This is performed for all frequencies and the filter parameters resulting in the lowest SSE are used.

3.2.6. Frequency Response Measurement

Each loudspeaker-room frequency response was obtained by taking a 4096-point Fast-Fourier Transform (FFT) of the loudspeaker-room impulse response. The impulse response was measured using maximum-length sequences [47].

3.2.6.1. Frequency Response Smoothing

The result of the FFT operation results in more data than required and the data points were linearly spaced in frequency. The data was converted to 1/x-octave measurements using a program that implemented the algorithm

described in [48]. Basically the algorithm performs power averaging using a sixth-order bandpass filter. The filter is swept in frequency at the required center frequency points and the output is normalized by the number of frequency points. When x=3, the algorithm implements third-octave measurements. Third-octave measurements can be justified on the basis of critical bandwidth theory since over a large part of the spectrum, third octave bandwidths closely approximate critical bandwidths [2]. However, sixth-octave measurements were generally made in this study for increased resolution. The process of measuring the frequency response and smoothing is summarized in Figure 3.15.

3.2.6.2. Target Frequency Response

An absolute target frequency response is determined from the measured frequency response. The target is influenced by such parameters as power handling capabilities of the loudspeaker at hand, its useful frequency range and subjective evaluation of the measured frequency response [49] - [53]. Also the target magnitude frequency response is not necessarily flat [1] [54]. The process of entering the target frequency response is summarized in Figure 3.16.

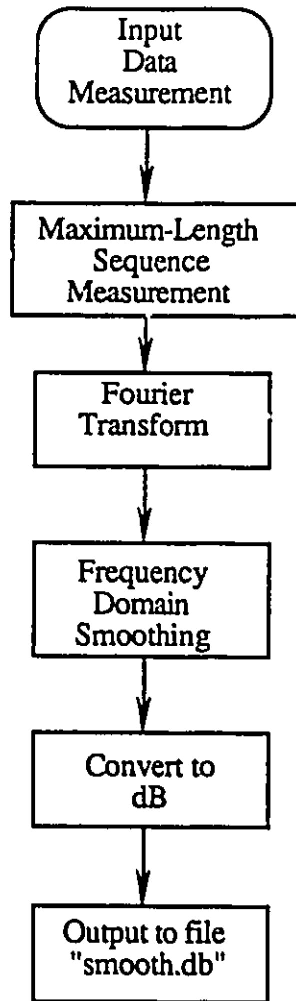


Figure 3.15 Input Data Measurement Algorithm

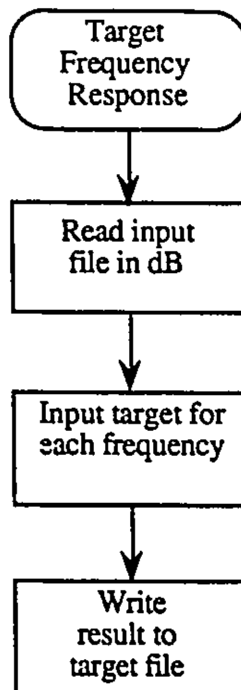


Figure 3.16 Target Frequency Response Algorithm

3.2.6.3. Normalization

There was an attempt made at normalizing the measured frequency response and the target frequency response. In this fashion, the target frequency response is specified with respect to a zero dB level instead of an absolute dBSPL level based on the measured response. Then the measured frequency response is normalized to zero dB for the maximum or minimum measured response (both were tried). This method allows the target response to be specified without attention to the absolute level. This method was used by Bullis and Brubaker [46]. The method performed poorly. The SSE was always larger than when the method without normalization was used.

3.2.7. Cascaded Versus Paralleled Sections

A cascaded topology has several advantages over a parallel topology. First, the total system response is easier to calculate since the total system log-magnitude response is simply the sum of the log-magnitude responses of all the stages. In other words, it is not a function of the phase of each stage. This is not the case for a parallel topology. This property simplifies the optimization procedure. This would not be a serious problem for a graphic equalizer since the center frequency and the Q of each stage are fixed. In the case of a parametric equalizer, the center frequencies can overlap such that parallel stages could interfere with one another.

Another benefit of cascaded sections is that if the response of each stage is minimum phase then the response of the total system is minimum phase. This is not necessarily the case for a parallel topology [7].

3.2.8. Global Automation Algorithm - Biquads

The global algorithm for finding a set of cascaded biquadratic equalizing filter will now be described. The global algorithm is shown in block diagram form in Figures 3.17a-c. First, the system frequency response is measured and the target frequency response is entered. The target equalization curve is found by taking the difference between the target and the system frequency responses. Note that all frequency responses are in units of dB. Then the center frequency where the peaks and dips occur are found through differentiation. Then the best K and Q are found for the first center frequency. The resulting SSE between this biquad stage and the target

equalization curve is found. This process is repeated for all center frequencies and the f_o, K and Q parameters resulting in the lowest SSE are used. The frequency response of this stage is then calculated. A new target equalization curve is computed by subtracting the response of the biquad stage from the old target equalization curve. This entire procedure is repeated until the desired number of stages is reached.

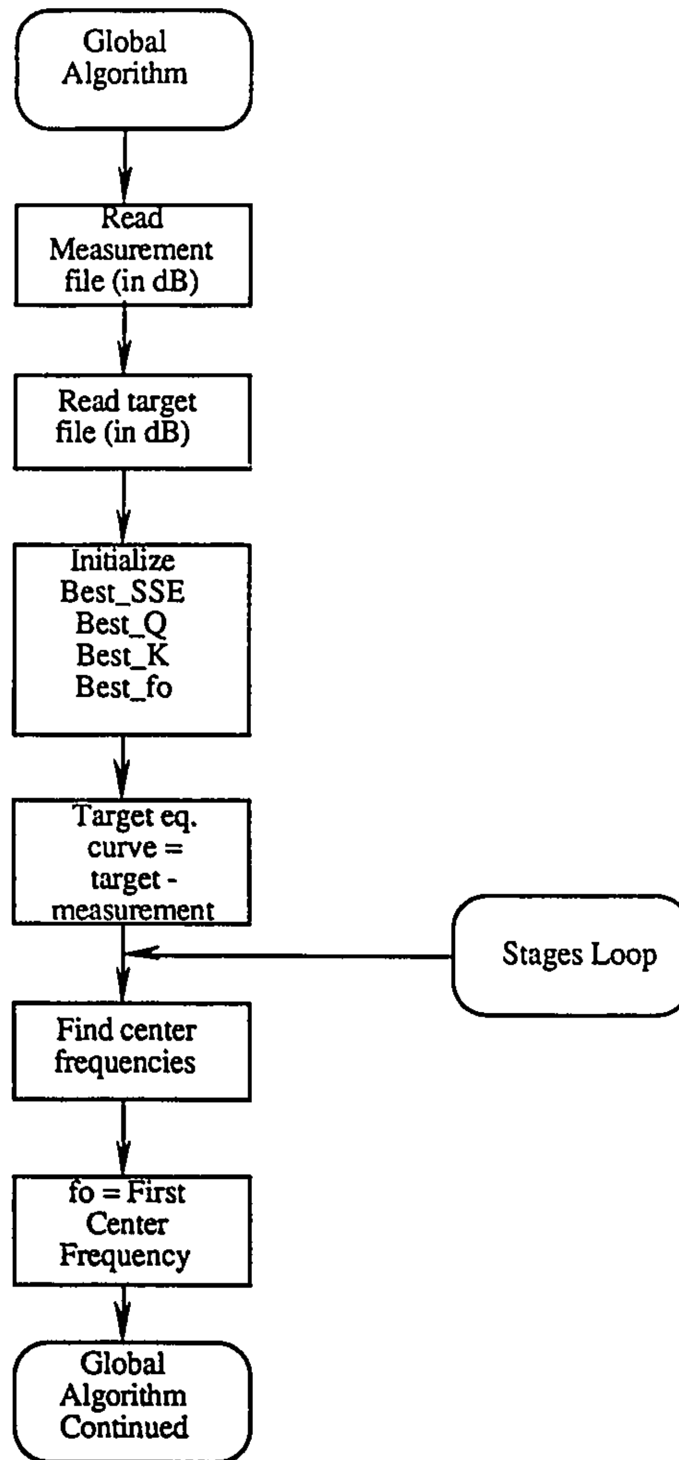


Figure 3.17a Global Algorithm

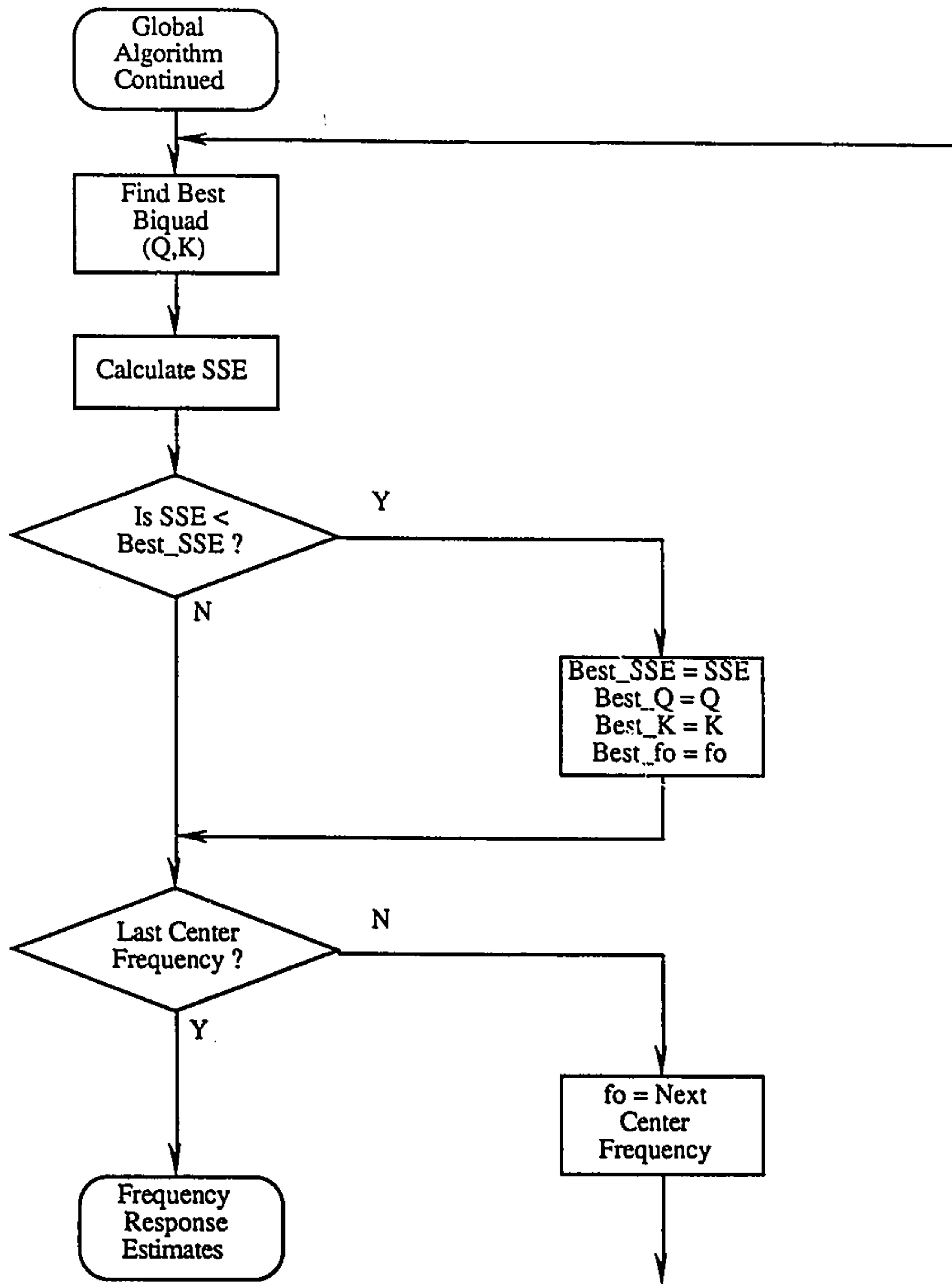


Figure 3.17b Global Algorithm

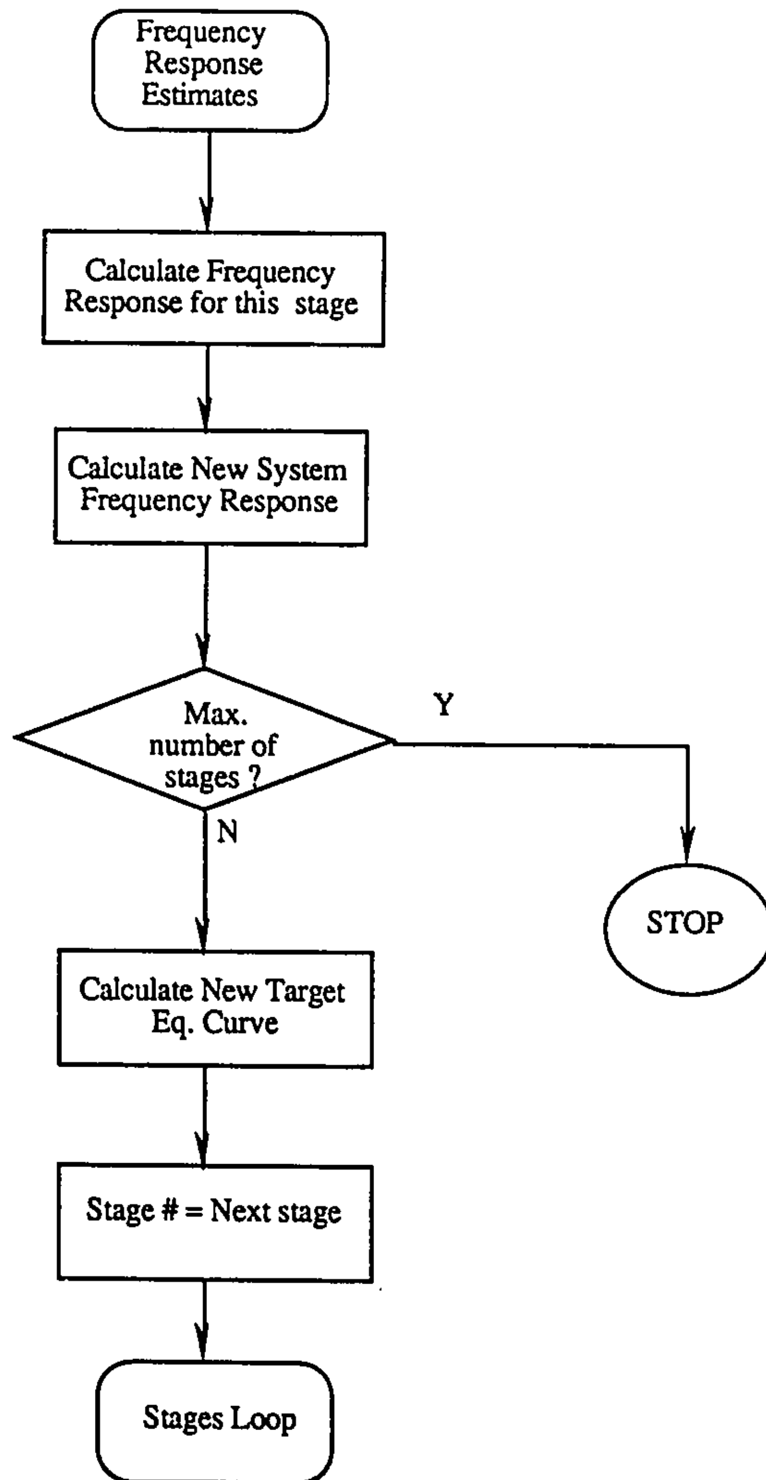


Figure 3.17c Global Algorithm

3.2.9. Global Automation Algorithm - Shelving

The global algorithm for finding the best shelving filter stages is identical to the one for the biquads except that automation method 2 for shelving filters described in section 3.2.5. is used instead. In addition, the center frequencies used are not only the peaks and dips but all the frequencies in the measurement.

3.2.10. Biquad Simulations

3.2.10.1. Subwoofer

Figure 3.18 shows the magnitude frequency response of a subwoofer in a listening room. This response was obtained by taking a 4096-point FFT of the subwoofer-room impulse response. Figure 3.19 shows the subwoofer-room magnitude response after 1/6-octave smoothing. The maximum peak-to-trough ratio is about 20 dB. Figure 3.20 shows the target frequency response together with the equalized magnitude frequency response. The equalized maximum peak-to-trough ratio is now about 9 dB and the response is much smoother. Eight biquadratic stages were optimized and the filter parameters are listed in Table 3.1. The exact response of each stage is shown in Figure 3.21 where six points per octave were used. The total response of the equalizer is shown in Figure 3.22. The optimization algorithm uses approximation (3.32) to calculate the equalizer response. Figure 3.23 shows the total equalizer response using this approximation. The difference is small and the error between the exact response and the approximation is shown in Figure 3.24. The error is within -0.4 dB and +0.1 dB for this case.

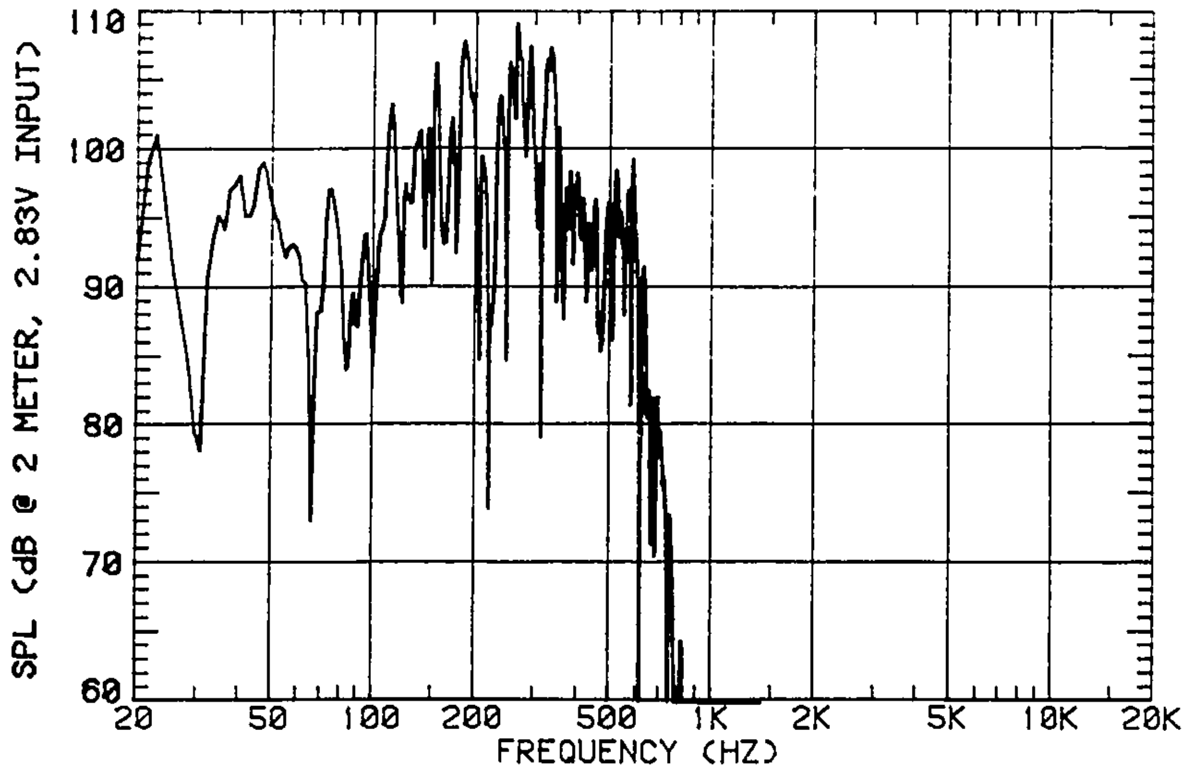


Figure 3.18 Subwoofer-Room Magnitude Response (FFT)

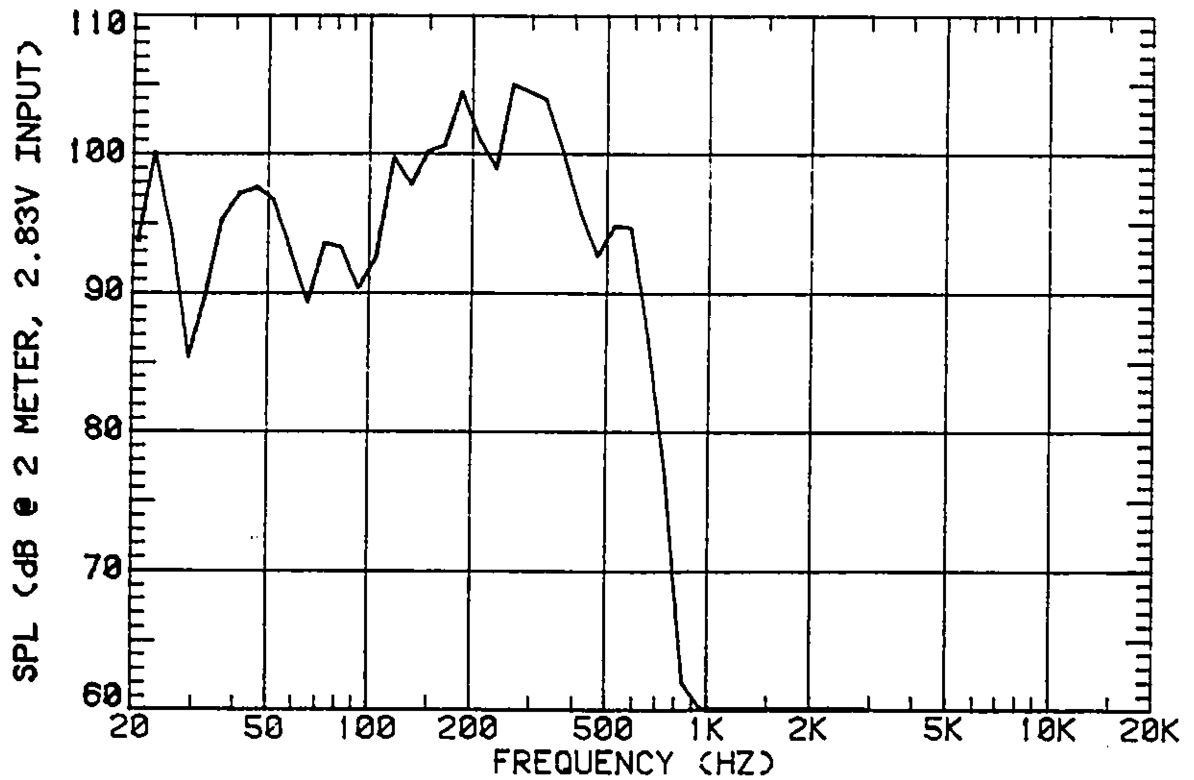


Figure 3.19 Subwoofer-Room Smoothed Magnitude Response

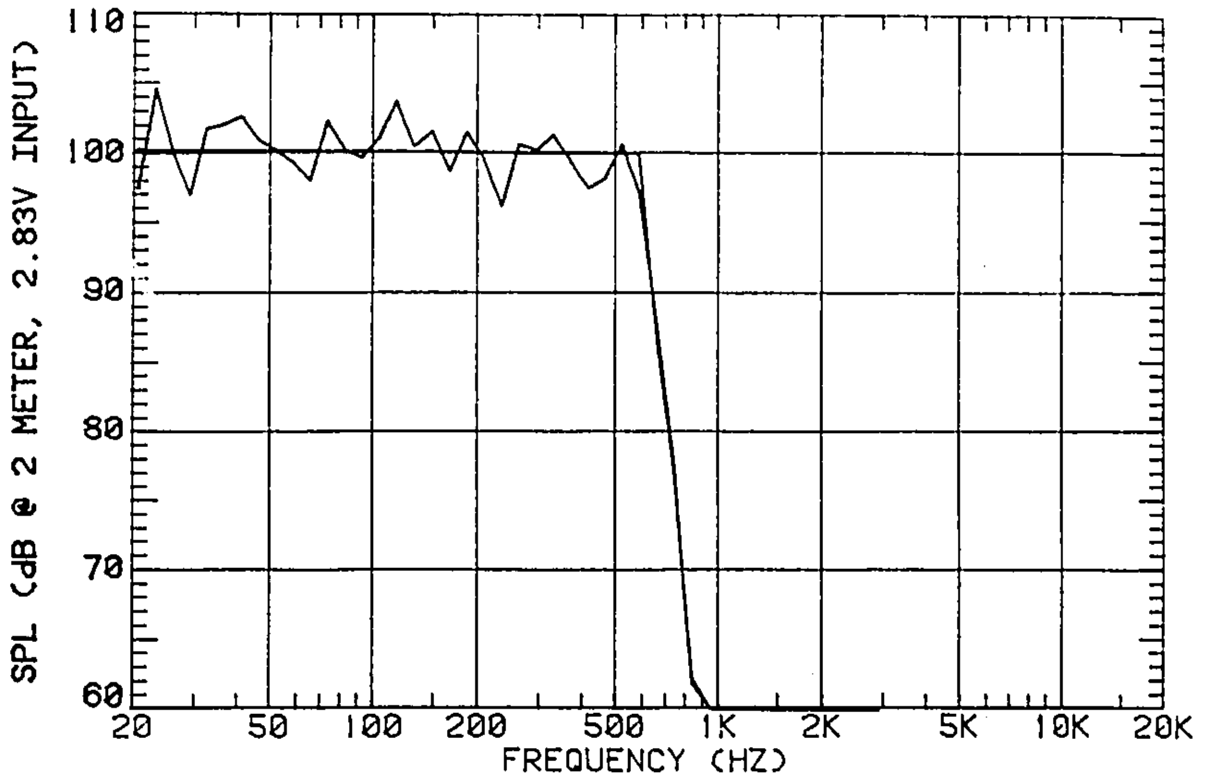


Figure 3.20 Subwoofer-Room Target and Equalized Magnitude Response

Stage #	f_o	Q	k	SSE
				1093
1	47.6	0.50	1.51	536
2	479.5	3.60	1.54	450
3	269.1	2.00	0.72	347
4	30.0	10.00	2.37	267
5	47.6	3.40	0.76	232
6	95.1	10.00	1.82	192
7	169.5	8.20	0.70	170
8	67.3	4.00	1.24	149

Table 3.1 Subwoofer Equalizer Filter Parameters

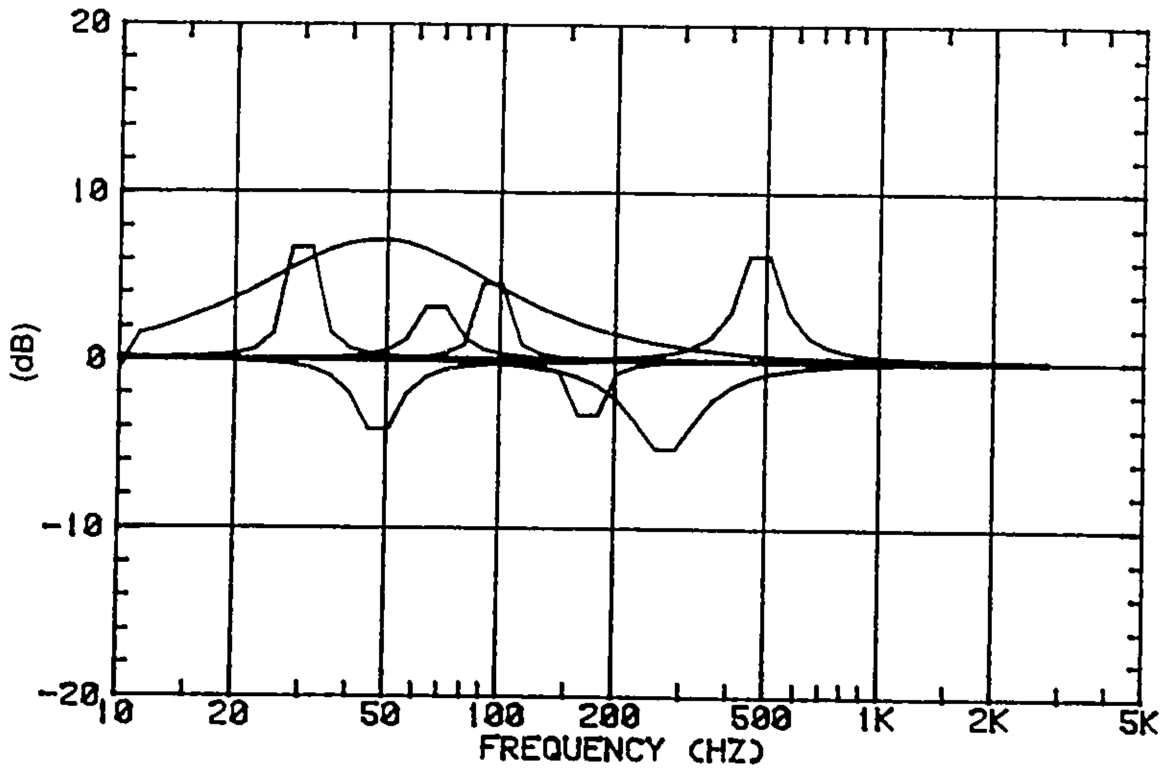


Figure 3.21 Exact Subwoofer Equalizer Response of Each Stage

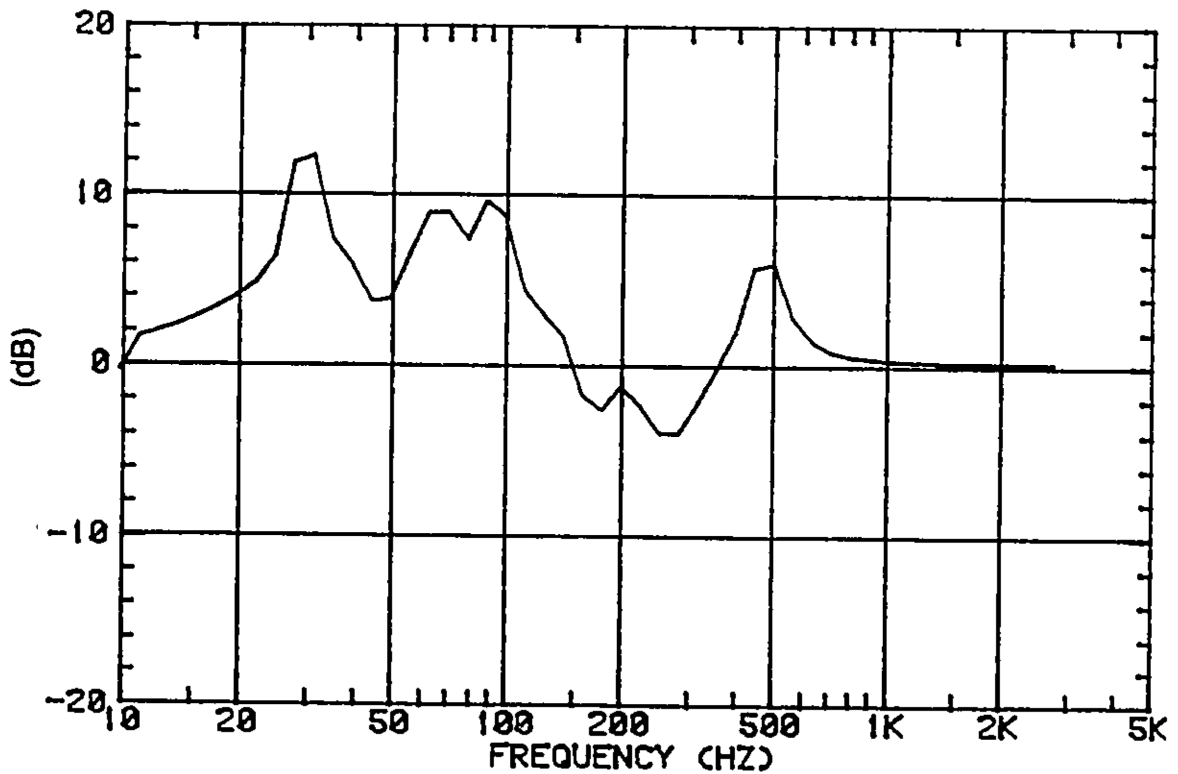


Figure 3.22 Exact Total Subwoofer Equalizer Response

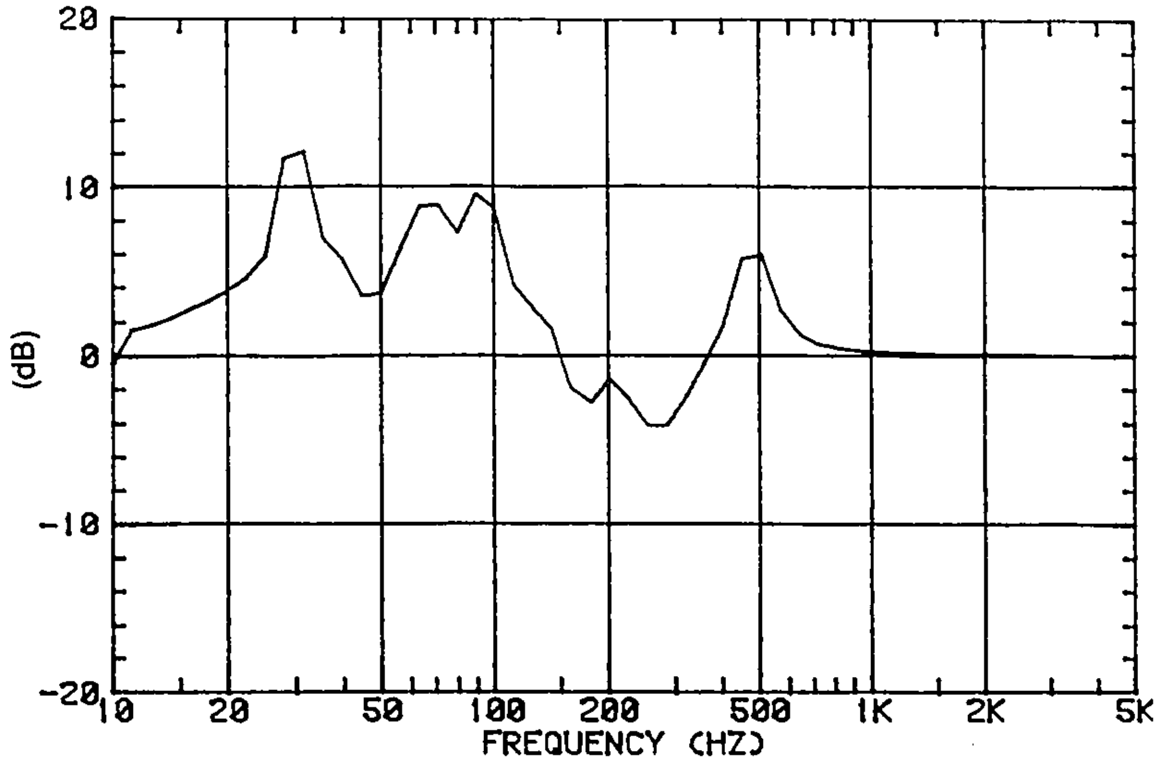


Figure 3.23 Approximate Total Subwoofer Equalizer Response

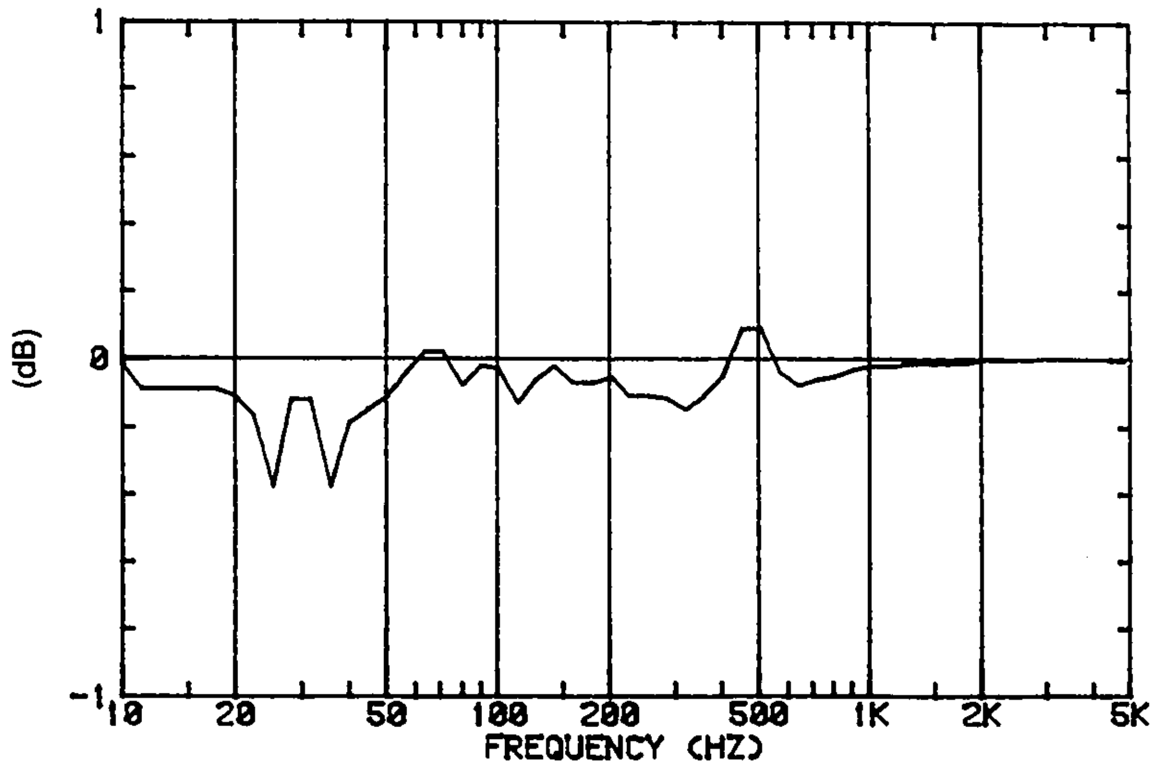


Figure 3.24 Total Subwoofer Equalizer Response Approximation Error

3.2.10.2. Satellite Speaker

A satellite speaker is one which is usually used with a subwoofer. A cross-over network allows the low frequencies to drive the subwoofer and the higher frequencies are fed to the satellite speaker. Figure 3.25 shows the 1/6-octave smoothed magnitude frequency response of a satellite speaker in a listening room. The signal to the satellite speaker was fed through a high-pass cross-over network. Figure 3.26 shows a target magnitude frequency response as well as an equalized frequency response. Seven biquadratic filters were optimized. The response of each one is shown in Figure 3.27 and the filter parameters are listed in Table 3.2. The total equalizer response is shown in Figure 3.28.

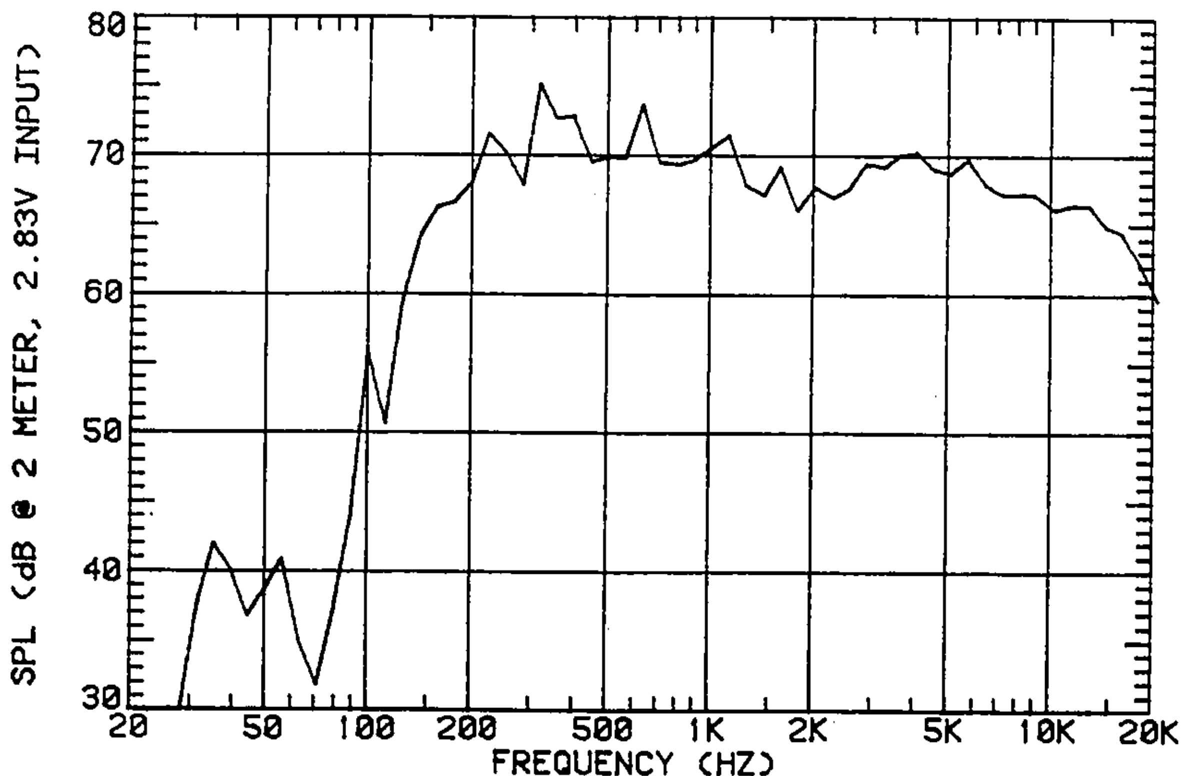


Figure 3.25 Satellite Speaker-Room Magnitude Response

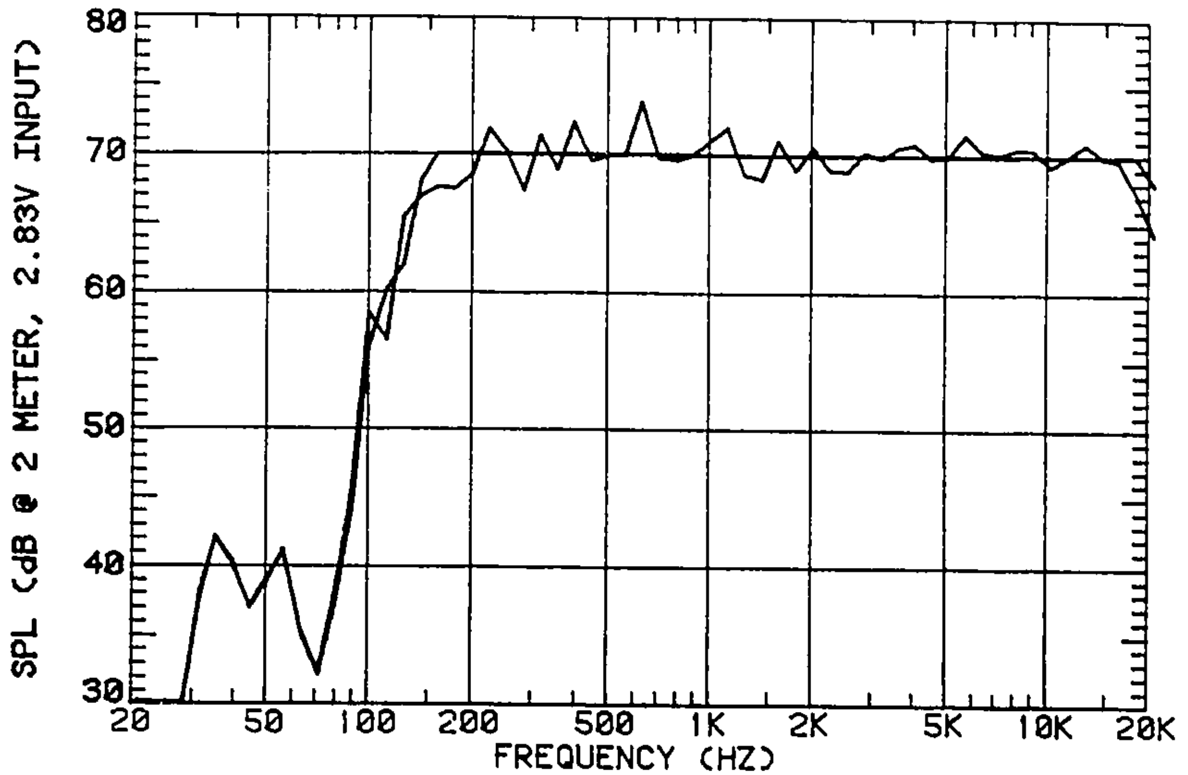


Figure 3.26 Satellite Equalized Speaker-Room Response and Target

Stage #	f_o	Q	k	SSE
				703
1	11680.6	0.60	1.35	436
2	115.0	3.60	1.47	354
3	325.2	10.00	0.58	320
4	1839.6	2.40	1.18	301
5	4635.4	0.80	0.90	281
6	13111.0	0.40	1.09	265
7	10406.2	1.80	0.84	241

Table 3.2 Satellite Equalizer Filter Parameters

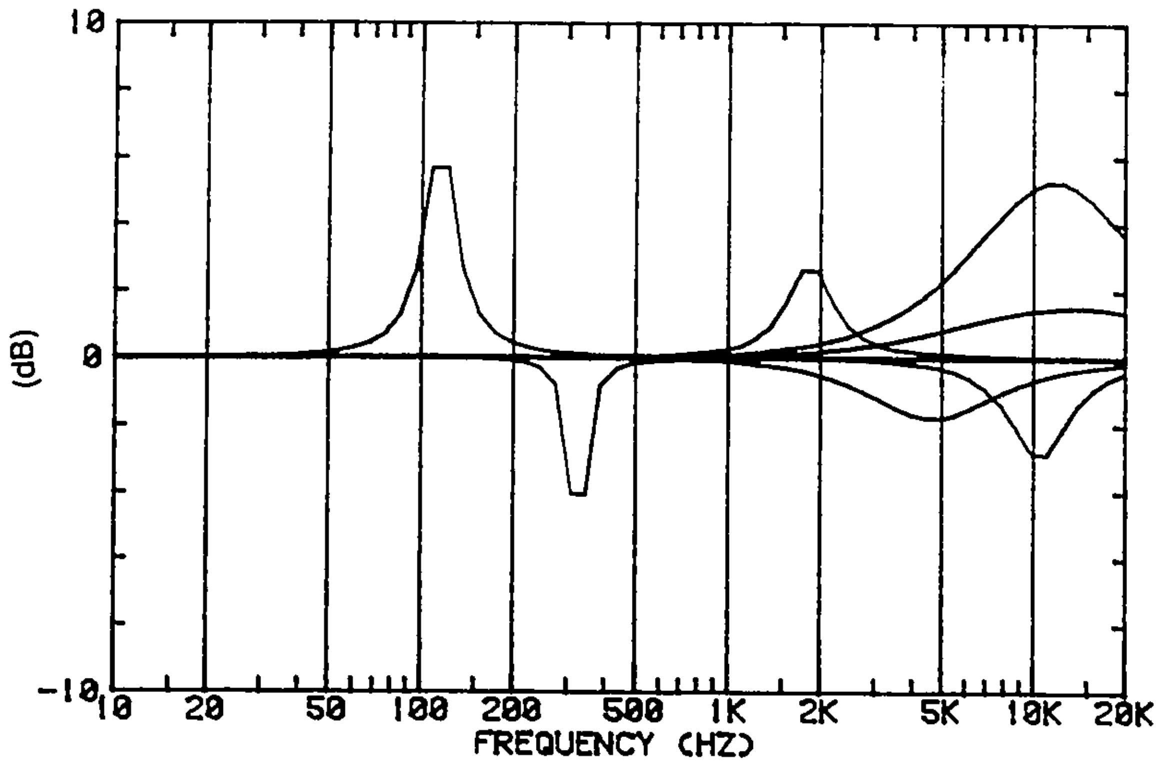


Figure 3.27 Exact Satellite Equalizer Response of Each Stage

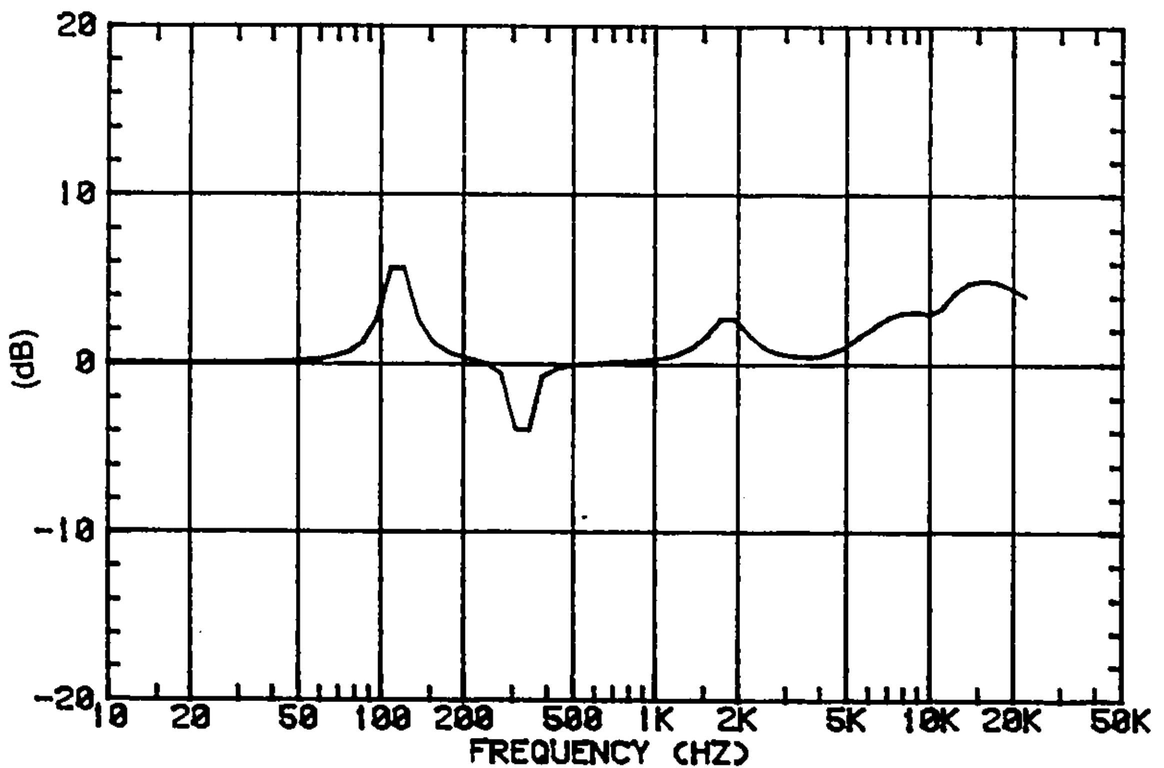


Figure 3.28 Exact Total Satellite Equalizer Response

3.2.11. Shelving Simulation

The subwoofer of Figure 3.19 was equalized using one shelving filter stage. The result is shown in Figure 3.29 along with the target response. Figure 3.30 displays the magnitude response of the shelving equalizer. The filter parameters are: f_o = 142.5 Hz and K = 1.29.

One problem with a first-order shelving filter is that its DC gain is not 1 (zero dB). This is unlike a second-order biquad where the gain returns to 0 dB at low frequencies. A loudspeaker is inherently a bandpass system. Therefore, a second-order equalizing filter whose magnitude response returns to zero dB at high and low frequencies - such that no equalization is performed in those regions - is well suited for this task. In practice, a shelving filter would need to be used with a high-pass filter, thereby increasing its order.

3.2.12. Subjective Tests

Subjective tests were not performed for the minimum-phase parametric equalizer since it is argued that in the absence of magnitude equalization, the phase is not altered. When magnitude equalization is performed, the system is minimum phase and this type of equalization is well documented and used in the audio field.

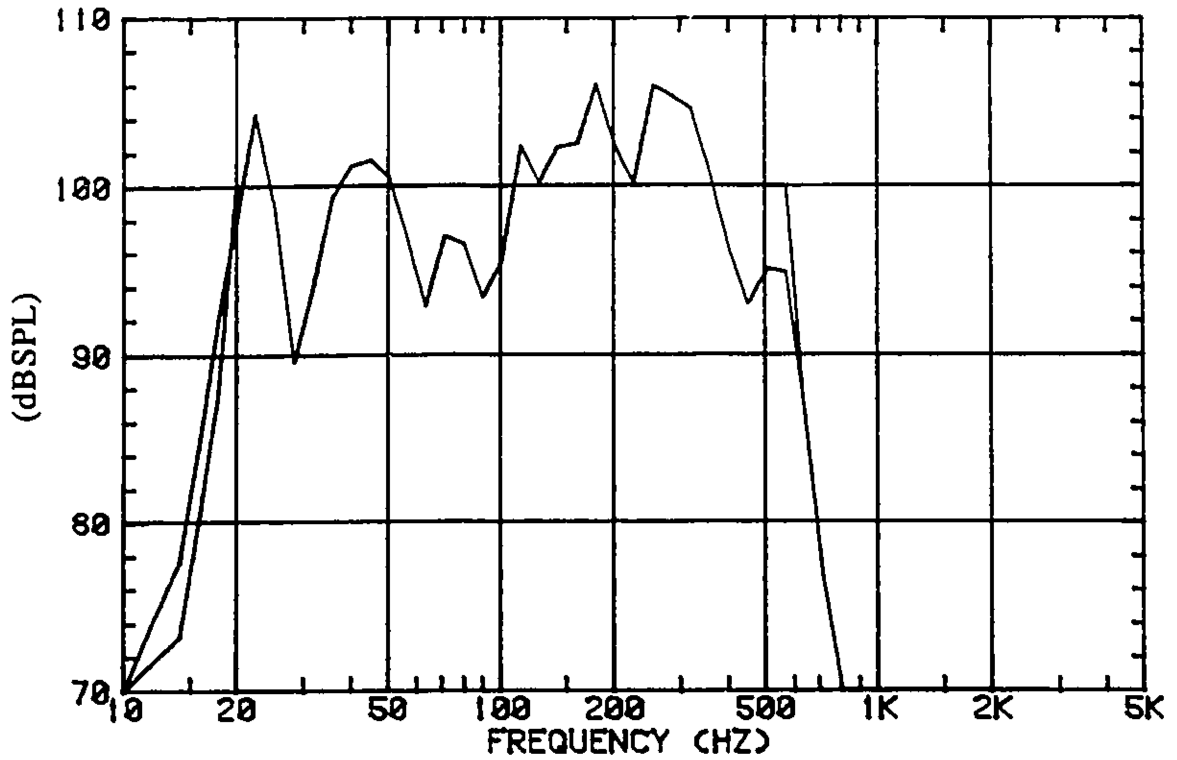


Figure 3.29 Equalized Subwoofer Response Using One Shelving Stage

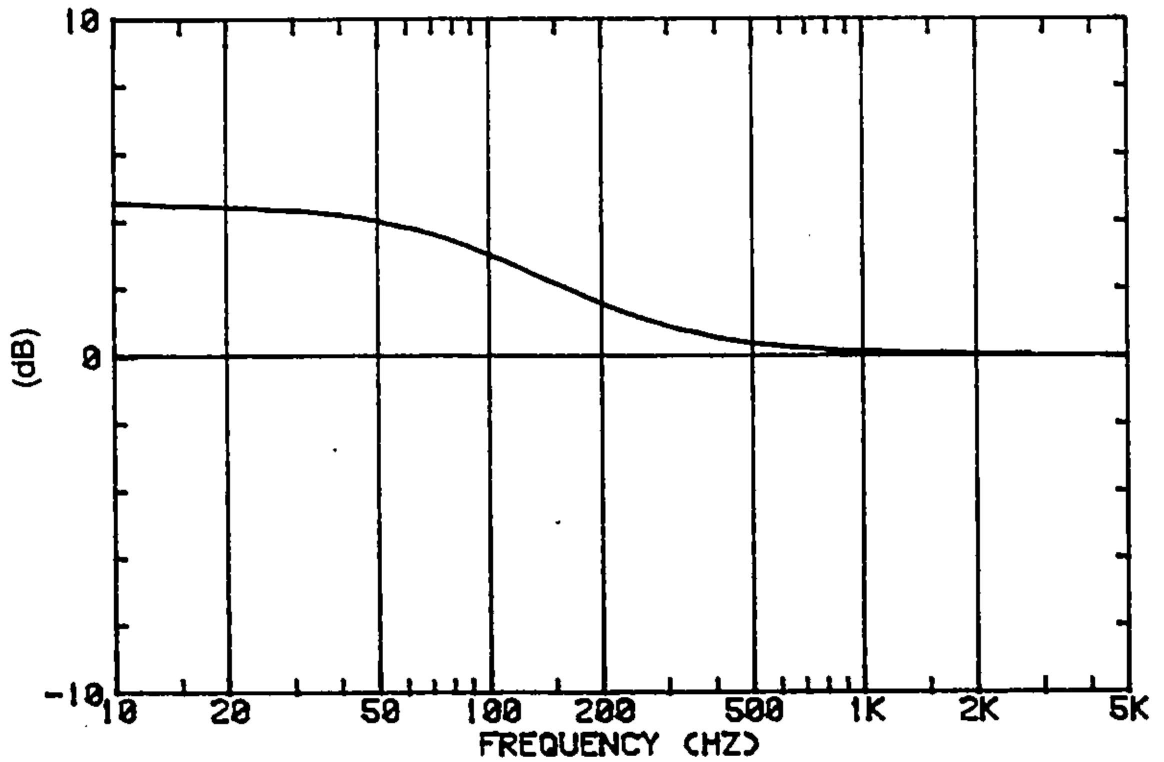


Figure 3.30 Shelving Subwoofer Equalizer Response

3.2.13. Digital Case

It was our aim to design digital filters with the same frequency response properties as the analog filters derived for the modified-Bode case. One method that guarantees a direct mapping between the analog and digital frequency responses is the bilinear transform [11] [22]. This transformation consists of substituting

$$s = \frac{2}{T}\,\frac{1 - z^{-1}}{1 + z^{-1}} \tag{3.57}$$

into the Laplace equation of the analog filter (where T is the sampling period). However, this transformation introduces a distortion of the frequency axis. This problem can be solved by pre-warping the analog frequency axis using

$$f_a = \frac{f_s}{\pi}\tan\left[\frac{\pi f_d}{f_s}\right] \tag{3.58}$$

where

f_a is the analog frequency in Hz

f_d is the digital frequency in Hz

and f_s is the sampling frequency in Hz (in our case 44100).

3.2.13.1. Frequency Warping Pre-Processing

The algorithms derived for the analog filters, can thus be used unchanged for digital filters, by simply pre-warping the measurement and target frequency axes using (3.58).

3.2.13.2. Digital Biquad Parameters

A digital biquadratic filter may have the following form

$$H_{BQ}(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2}}{1 + b_1 z^{-1} + b_2 z^{-2}} \tag{3.59}$$

Substituting (3.57) into (3.7), generates the following digital biquadratic filter parameters

$$a_0 = \left[4 + \frac{2A\omega_o T}{Q} + \omega_o^2 T^2\right] / C \tag{3.60a}$$

$$a_1 = 2\left[\omega_o^2 T^2 - 4\right] / C \tag{3.60b}$$

$$a_2 = \left[4 - \frac{2A\omega_o T}{Q} + \omega_o^2 T^2\right] / C \tag{3.60c}$$

$$b_1 = a_1 \tag{3.60d}$$

$$b_2 = \left[4 - \frac{2B\omega_o T}{Q} + \omega_o^2 T^2\right] / C \tag{3.60e}$$

where

$$C = 4 + \frac{2B\omega_0 T}{Q} + \omega_0^2 T^2 \tag{3.60f}$$

and in the case of a modified-Bode equalizing biquad filter.

$$A = k$$

$$B = 1/k.$$

3.2.13.3. Digital Shelving Parameters

A digital shelving filter may have the following form

$$H_s(z) = \frac{a_0 + a_1 z^{-1}}{1 + b_1 z^{-1}} \tag{3.61}$$

Substituting (3.57) into (3.39), generates the following digital shelving filter parameters

$$a_0 = (\omega_z T + 2)/C \tag{3.62a}$$

$$a_1 = (\omega_z T - 2)/C \tag{3.62b}$$

$$b_1 = (\omega_p T - 2)/C \tag{3.62c}$$

where

$$C = \omega_p T + 2 \tag{3.62d}$$

and in the case of a modified-Bode shelving filter

$$\omega_z = K\omega_o$$

$$\omega_p = \omega_o/K.$$

3.2.13.4. Biquad Simulation

The same satellite speaker whose frequency response is shown in Figure 3.25 was equalized using digital biquadratic filters. The same target frequency response was used (Figure 3.26). Those responses were pre-warped and the results are shown in Figure 3.31. The equalized frequency response is shown in Figure 3.32. Seven biquadratic filters were optimized. The filter parameters are listed in Table 3.3 and the response of each one is shown in Figure 3.33. The total equalizer response is shown in Figure 3.34.

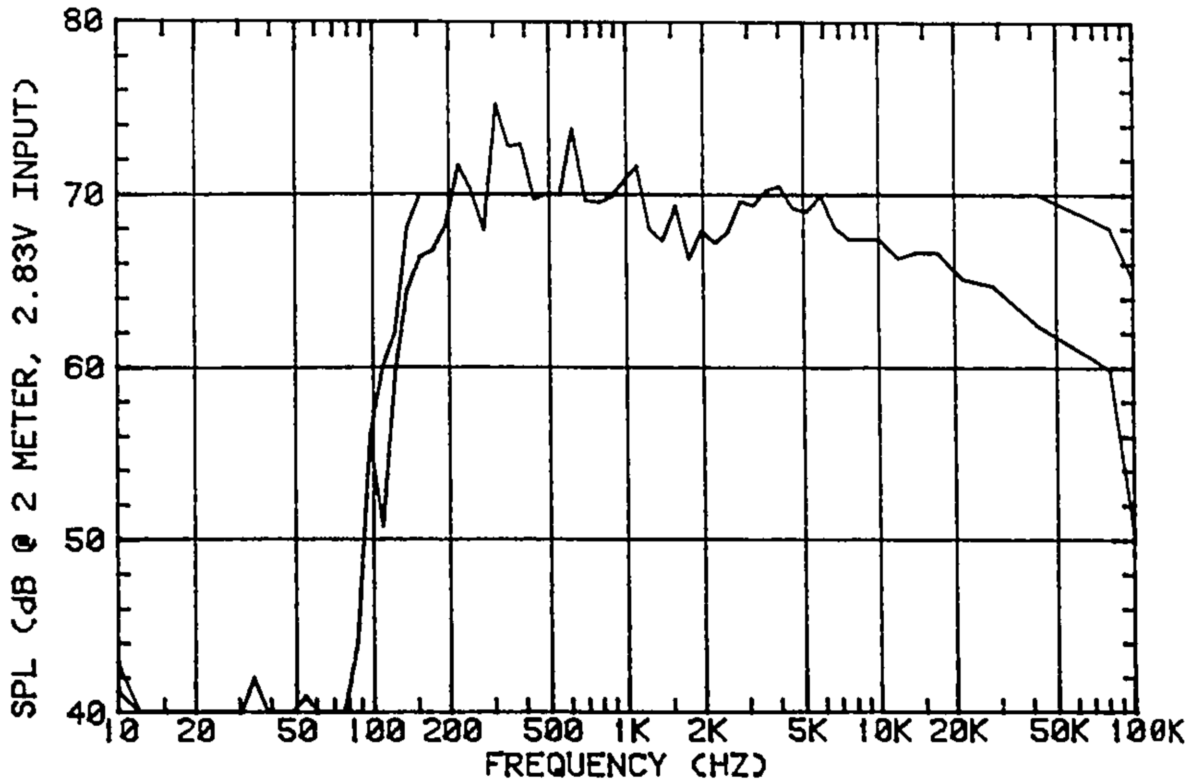


Figure 3.31 Pre-Warped Satellite and Target Responses

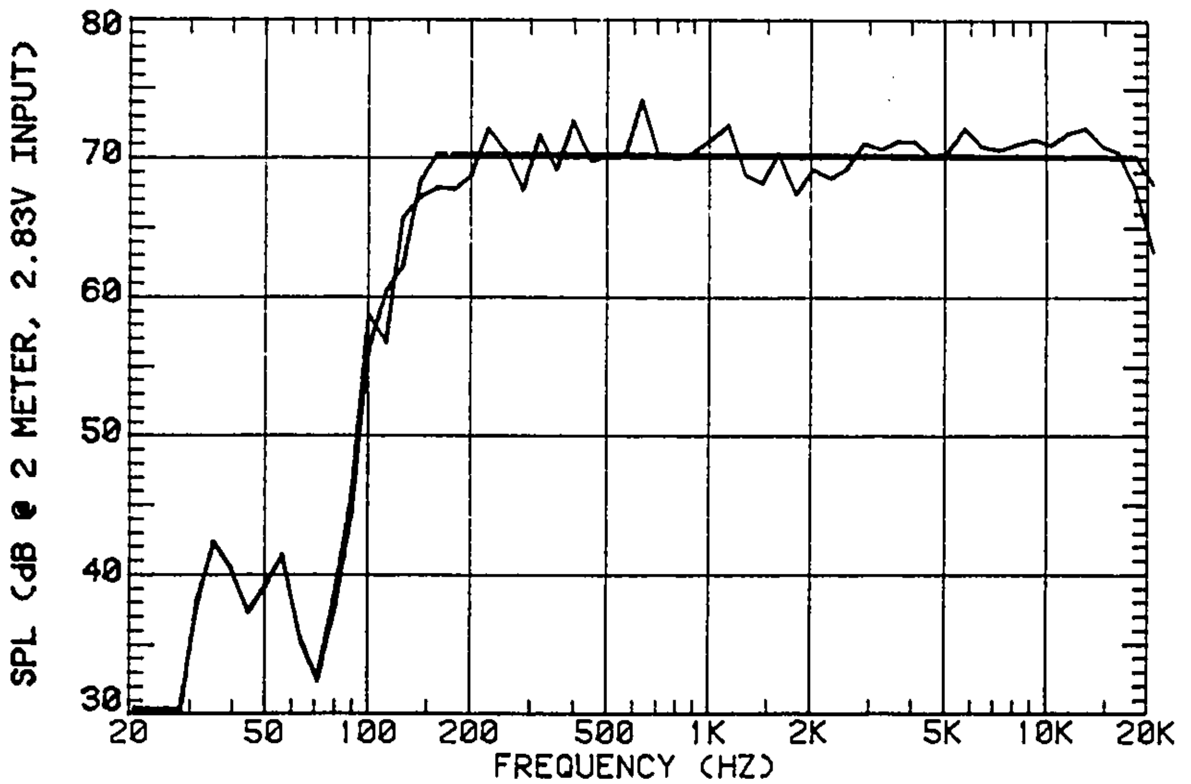


Figure 3.32 Digitally Equalized Satellite Speaker Response

Stage #	f_o	Q	k	SSE
				703
1	15463.9	0.20	1.23	507
2	115.0	3.60	1.47	425
3	325.3	10.00	0.58	391
4	4254.0	1.40	0.86	367
5	19078.7	0.20	1.09	338
6	8122.1	0.80	0.90	320
7	19078.7	0.20	1.06	303

Table 3.3 Digital Satellite Equalizer Filter Parameters

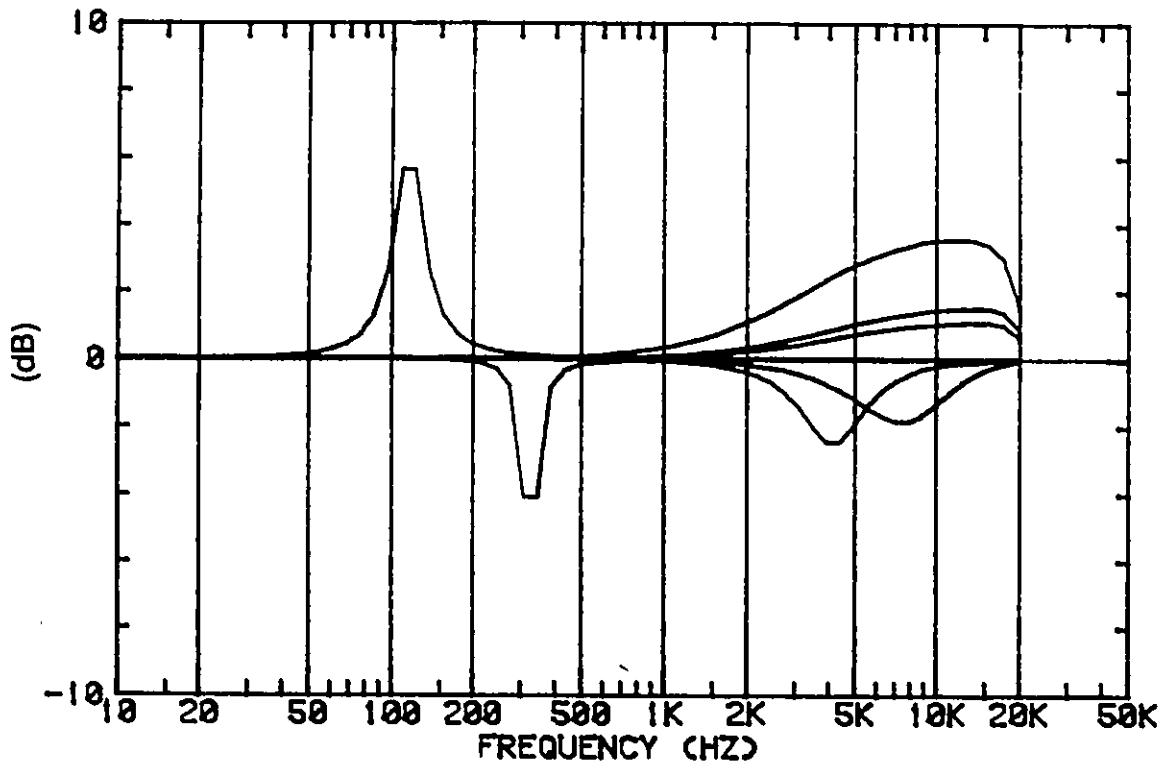


Figure 3.33 Exact Digital Equalizer Response of Each Stage

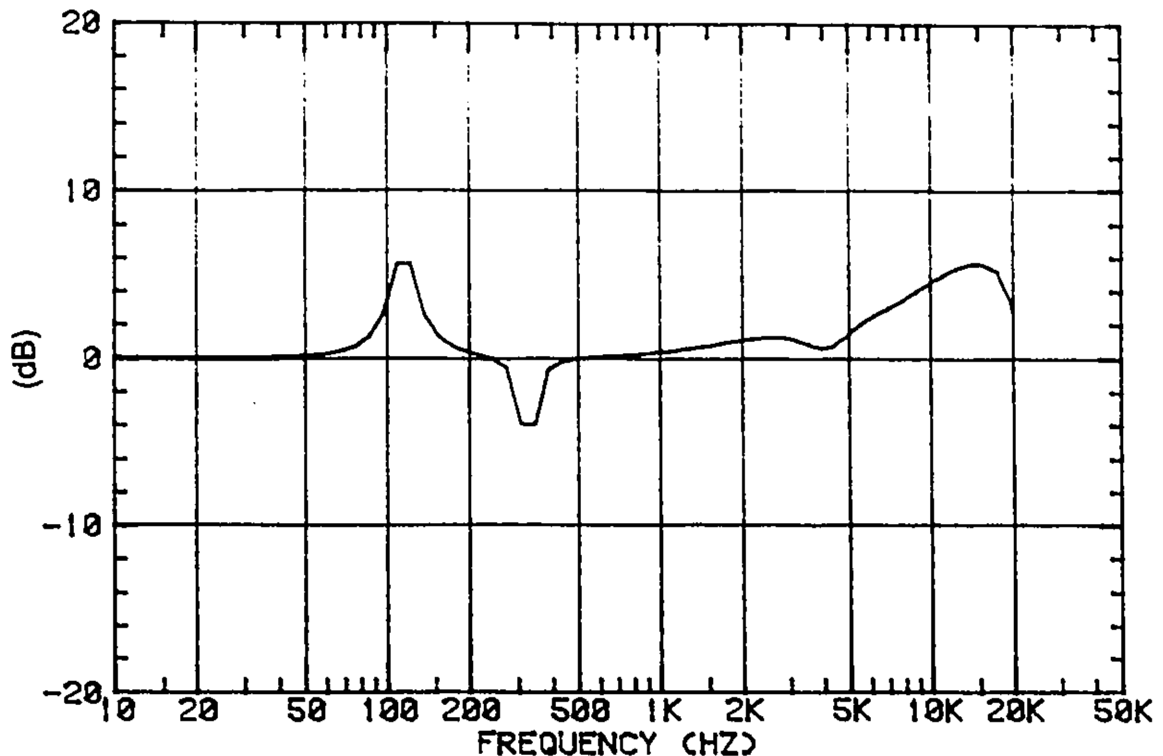


Figure 3.34 Exact Total Digital Satellite Equalizer Response

Notice that the filter parameters for the low frequency stages are similar to the analog equalization case (compare Table 3.3 and Table 3.2). At higher frequencies, the parameters differ due to the increased effect of frequency warping.

3.2.13.5. Shelving Simulation

The subwoofer response shown in Figure 3.19 was used to perform a digital shelving equalizer simulation. The exact same filter parameters as in the analog case were obtained since the subwoofer response is limited to low frequencies where warping is negligible.

Chapter 4

4. Conclusions and Recommendations

4.1. Conclusions

Non-minimum phase filters and equalizers must be used with caution in high-fidelity audio since it is possible to effect a subjective timbral modification of some signals even when the magnitude frequency response of the signal is not modified.

A new type of equalizing filter has been introduced where the log-magnitude response is approximately linear in the gain term over a wide range of boost and cut. In addition, the log-magnitude response of this filter can be approximated using a simple algebraic equation.

A method was presented by means of which the magnitude frequency response of a loudspeaker in a listening room can be automatically equalized to meet some target magnitude response. The method uses an iterative method to find the approximate optimum Q and center frequency of a biquadratic equalizing filter. Then the gain term is found using a direct best least-squares formula. Many stages can be optimized in succession such that the cascade of

all the stages and the measurement frequency response approximates some target frequency response.

An optimizing method based on a paper by Bullis and Brubaker [46] failed to converge to a best least-squares solution even for simple frequency response curves. It was found that this method fails in part because it modifies the least-squares formulation such that it is no longer representative of the actual error that must be minimized. The method converges to the correct solution when exact data is used but not for the general case.

It was found that optimizing second-order stages one at a time requires the use of a frequency weighting curve. Otherwise, the best least-squares solution is often a flat gain stage. The optimization procedure must be forced to operate on only a portion of the frequency response such that each anomaly is dealt with in succession.

A normalization scheme was tried whereby the measured frequency response is normalized with respect to its maximum or minimum point. This method failed to yield satisfactory results since it effectively increased the required system order unnecessarily by forcing boost curves to match where cut curves should be used and vice versa. The best results were obtained when a target frequency response was specified in absolute dBSPL based on the measured response.

First-order shelving filters are sometimes used in the audio field. These filters must be used with caution since low frequencies outside the useful range of a

loudspeaker may be boosted beyond its power handling capability. In practice, these filters must be bandlimited - thereby increasing their effective order. Second-order equalizing filters do not have this problem since their magnitude frequency response naturally returns to 0 dB outside the audio band.

It was found that the use of K as a scaling factor for the zeros together with $1/K$ as the scaling factor for the poles of a biquadratic equalizing filter results in optimum linearity in the gain term of the log-magnitude response.

4.2. Recommendations for Further Work

The application of the modified-Bode equalizing filter to graphic equalizers should be investigated. Specifically, the investigation should compare interband interference and magnitude response ripple behavior with that of constant-Q designs.

Post-processing optimization techniques for the modified-Bode parametric equalizer should be investigated. It may be possible to develop a technique whereby the filter parameters found by the method introduced in this document are used as a starting point and then optimized using a non-linear optimization technique such that the target frequency response is better approximated.

The automatic adjustment procedure described in this work optimizes one stage at a time. This may result in more stages than required. It may be possible to develop a technique where several stages are optimized at the same time. Alternatively, the response of two or more stages may be approximated by one stage.

The optimum search grid resolution on the parameter Q should be investigated. A high resolution is wasteful of computing power and is not necessary. A low resolution on Q results in additional stages required.

Non-linear optimization techniques should be investigated to find Q and f_o. Such a technique may save some computing time over the iterative method used in this work.

An algorithm could easily be developed where a desired maximum ripple is specified and the algorithm would add biquad stages until the ripple specification is met.

Fixed-point digital implementation issues regarding IIR equalizing filters as applied to high-fidelity audio should be investigated. The reader is referred to [55] as a good starting point.

Finally, the equations and algorithms developed in this work should be implemented in hardware.

References

[1] W. R. Bevan, "A Simplified Equalization Analyzer", *J. Audio Eng. Soc.*, Vol. 26, No. 3, pp. 120-129, March 1978.

[2] John Eargle, "Equalizing the Monitoring Environment", *J. Audio Eng. Soc.*, Vol. 21, No. 2, pp. 103-107, March 1973.

[3] Glen Ballou, editor, *Handbook for Sound Engineers - The New Audio Cyclopedia*, Howard W. Sams & Company, Indianapolis, Indiana, 1987, pp. 569-612, pp. 1028-1039.

[4] Don Davis and Carolyn Davis, *Sound System Engineering*, sec. ed., Howard W. Sams & Company, Indianapolis, Indiana, 1987, pp 423-469.

[5] O. J. Bonello, "Modular Parametric Equalizer-Filter, A New way to Synthesize the Frequency Response", *Audio Eng. Soc.* Preprint #1170, presented at the 55th Convention, Oct. 29 - Nov. 1, 1976.

[6] Dennis A. Bohn, "Constant-*Q* Graphic Equalizers", *J. Audio Eng. Soc.*, Vol. 34, No. 9, pp. 611-626, September 1986.

[7] R. A. Greiner and Micheal Schoessow, "Design Aspects of Graphic Equalizers", *J. Audio Eng. Soc.*, Vol. 31, No. 6, pp. 394-407, June 1983.

[8] Paul H. Kraght, "A Linear-Phase Digital Equalizer with Cubic-Spline Frequency Response", *Audio Eng. Soc.* Preprint #2971, presented at the 89th convention, Sept. 21-25, 1990.

[9] Terry Pennington, "Constant Q", *Studio Sound*, Vol. 27,No. 10, pp. 82-85, October 1985.

[10] Richard C. Heyser, "Loudspeaker Phase Characteristics and Time Delay Distortion: Part 1", *J. Audio Eng. Soc.*, Vol. 17, No. 1, pp. 30-41, January 1969.

[11] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*, p. 346, Englewood Cliffs, NJ: Prentice-Hall, 1975.

[12] Ragnar Hergum, "A Low Complexity, LinearPhase Graphic Equalizer", *Audio Eng. Soc.* Preprint #2938, presented at the 85th convention, Nov. 3-6, 1988.

[13] Jorgen Arendt Jensen, "A New Principle for an All-Digital Preamplifier and Equalizer", *J. Audio Eng. Soc.*, Vol. 35, No. 12, pp. 994-1003, December 1987.

[14] Mark Erne, Christof Heidelberger, "Design of a DSP-Based 27 Band Digital Equalizer", *Audio Eng. Soc.* Preprint #3029, presented at the 90th Convention, Feb. 19-22, 1991.

[15] Juan A. Hendriquez, Terry E. Riemer and Russell E. Trahan, Jr., "A Phase-Linear Audio Equalizer: Design and Implementation", *J. Audio Eng. Soc.*, Vol. 38, No. 9, pp. 653-666, April 1990.

[16] T. E. Riemer and R. E. Trahan, "Design of a Phase-Linear Digital Audio Equalizer", *Audio Eng. Soc.* Preprint #2045, presented at the 74th Convention, Oct. 8-12, 1983.

[17] James A. Moorer and Mark Berger, "Linear-Phase Bandsplitting: Theory and Applications", *J. Audio Eng. Soc.*, Vol. 34, No. 3, pp. 143-152, March 1986.

[18] W. F. McGee, "Design of Digital Filter Banks with Allpass Reconstruction Property", Department of Electrical Engineering, University of Ottawa, 1990.

[19] Motorola, "Digital Stereo 10-Band Graphic Equalizer Using the DSP56001", Motorolla Application Note, 1988.

[20] R. E. Crochiere and L. R. Rabiner, *Multirate Digital Signal Processing*, p. 289, Englewood Cliffs, NJ: Prentice-Hall, 1983.

[21] Martin Thomas, "Tunable Audio Equalizer", *Wireless World*, September 1978.

[22] James A. Moorer, "The Manifold Joys of Conformal Mapping: Applications to Digital Filtering in the Studio", *J. Audio Eng. Soc.*, Vol. 31, No. 11, pp. 826-841, November 1983.

[23] Stanley A. White, "Design of a Digital Biquadratic Peaking or Notch Filter for a Digital Equalizer", *J. Audio Eng. Soc.*, Vol. 34, No. 6, pp. 479-483, June 1986.

[24] P. A. Regalia and S. K. Mitra, "Tunable Digital Frequency Response Equalization Filters", *IEEE Trans. Acous., Speech and Sig. Proc.*, Vol. ASSP-35, No. 1, pp. 118-120, pp. 118-120, January 1987.

[25] Colin Bean and Peter Craven, "Loudspeaker and Room Correction Using Digital Signal Processing", *Audio Eng. Soc.* Preprint #2756, presented at the 86th convention, Mar. 7-10, 1989.

[26] S. J. Elliott and P. A. Nelson, "Multiple-Point Equalization in a Room Using Adaptive Digital Filters", *J. Audio Eng. Soc.*, Vol. 37, No. 11, pp. 899-907, November 1989.

[27] J. Kuriyama and Y. Furukawa, "Adaptive Loudspeaker System", *J. Audio Eng. Soc.*, Vol. 37, No. 11, pp. 919-926, November 1989.

[28] S. T. Neely and J. B. Allen, "Invertibility of a Room Impulse Response", *J. Acoust. Soc. Am.*, 66 (1), pp.165-169, July 1979.

[29] W.F. McGee, "Frequency Interpolation Filter Bank", *ISCAS* , pp. 1563-1566, 1989.

[30] Gabor Peceli, "Resonator-Based Digital Filters", *IEEE Transactions on Circuit and Systems*, Vol. 36, No. 1, pp. 156-159, January 1989.

[31] Gabor Peceli, "Sensitivity Properties of Resonator-Based Digital Filters", *IEEE Transactions on Circuit and Systems*, Vol. 35, No. 35, pp. 1195-1197, September 1988.

[32] W. F. McGee and Genzao Zhang, "Resonator-Based Logarithmic Filter Banks", Department of Electrical Engineering, University of Ottawa, 1990.

[33] Stanley P. Lipshitz, Mark Pocock and John Vanderkooy, "On the Audibility of Midrange Phase Distortion in Audio Systems", *J. Audio Eng. Soc.*, Vol. 30, No. 9, pp. 580-595, September 1982.

[34] D. Preis, "Measures and Perceptions of Phase Distortion in Electroacoustical Systems", *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing* (Denver, CO, 1980 Apr. 9-11), vol. 2. pp. 490-493 (1980).

[35] D. Preis, "Linear Distortion", *J. Audio Eng. Soc.*, Vol. 24, No. 5, pp. 346-367, June 1976.

[36] J. A. Deer, P. J. Bloom and D. Preis, "Perception of Phase Distortion in All-Pass Filters", *J. Audio Eng. Soc.*, Vol. 33, No. 10, pp. 782-786, October 1985.

[37] D. Preis, "Phase Distortion and Phase Equalization in Audio Signal Processing - A Tutorial Review", *J. Audio Eng. Soc.*, Vol. 30, No. 11, pp. 774-794, November 1982.

[38] D. Preis and P. J. Bloom, "Perception of Phase Distortion In Anti-Alias Filters", *J. Audio Eng. Soc.*, Vol. 32, No. 11, pp. 842-848, November 1984.

[39] Villy Hansen and Erick Rorbaek Madsen, "On Aural Phase Detection", *J. Audio Eng. Soc.*, Vol. 22, No. 1, pp. 10-14, Jan./Feb. 1974.

[40] Villy Hansen and Erick Rorbaek Madsen, "On Aural Phase Detection: Part II", *J. Audio Eng. Soc.*, Vol. 22, No. 10, pp. 783-788, December 1974.

[41] H. W. Bode, "Attenuation Equalizer", U.S. Patent No. 2,096,027, October 19, 1937.

[42] W. R. Lundry, "Wave Transmission Network", U.S. Patent No. 2,070,668, February 16, 1937.

[43] E. L. Norton, "Attenuation Equalizer", U.S. Patent No. 2,019,624, November 5, 1935.

[44] H. W. Bode, "Variable Equalizers", *Bell System Technical Journal*, Vol. 17, pp. 229-244, April 1938.

[45] Franc Brglez, "Inductorless Variable Equalizers", *IEEE Trans. on Circuits and Systems*, Vol. CAS-22, No. 5, pp. 415-419, May 1975.

[46] D. C. Bullis and T. A. Brubaker, "An Algorithm for Estimation of Parametric Filter Functions", *J. Audio Eng. Soc.*, Vol. 29, No. 4, pp. 235-242, April 1981.

[47] Douglas D. Rife and John Vanderkooy, "Transer-Function Measurement with Maximum-Length Sequences", *J. Audio Eng. Soc.*, Vol. 37, No. 6, pp. 419-444, June 1989.

[48] S. P. Lipshitz, T. C. Scott and J. Vanderkooy, "Increasing the Audio Measurement Capability of FFT Analyzers by Microcomputer Post-Processing", *Audio Eng. Soc.* Preprint #2050, presented at the 74th Convention, Oct. 8-12, 1983.

[49] Floyd E. Toole, "Listening Tests - Turning Opinion Into Fact", *J. Audio Eng. Soc.*, Vol. 30, No. 6, pp. 431-445, June 1982.

[50] Floyd E. Toole, "Subjective Measurements of Loudspeaker Sound Quality and Listener Performance", *J. Audio Eng. Soc.*, Vol. 33, No. 1/2, pp. 2-32, Jan./Feb.1985.

[51] Floyd E. Toole, "Loudspeaker Measurements and Their Relationship to Listener Preferences: Part 1", *J. Audio Eng. Soc.*, Vol. 34, No. 4, pp. 227-235, April 1986.

[52] Floyd E. Toole, "Loudspeaker Measurements and Their Relationship to Listener Preferences: Part 2", *J. Audio Eng. Soc.*, Vol. 34, No. 5, pp. 323-348, May 1986.

[53] Floyd E. Toole and Sean E. Olive, "The Modification of Timbre by Resonances: Perception and Measurement", *J. Audio Eng. Soc.*, Vol. 36, No. 3, pp. 122-142, March 1988.

[54] Roy F. Allison and Robert Berkovitz, "The Sound Field in Home Listening Rooms", *J. Audio Eng. Soc.*, Vol. 20, No. 6, pp. 459-469, July/Aug. 1972.

[55] Jon Dattorro, "The Implementation of Recursive Digital Filters for High-Fidelity Audio", *J. Audio Eng. Soc.*, Vol. 36, No. 11, pp. 851-878, November 1988.

Appendix

Software Listings

Appendix A - Filterbank Frequency Response

```
/*****************************************************************/
/*  file trans20.c                                                */
/*                                                                */
/*  Same as transl2.c with a try at a more modular approach      */
/*  with some filter parameter include files.                     */
/*                                                                */
/* This program assumes that the sum of the r's equals 1.        */
/* Transfer function implemented here is based on :              */
/*                                                                */
/* "Design of Digital Filterbanks with Allpass                   */
/*  Reconstruction Property"                                      */
/*                                                                */
/*              W. F. McGee                                       */
/*    Department of Electrical Engineering                        */
/*          University of Ottawa                                  */
/*                                                                */
/*                                                                */
/*                                                                */
/* Analytic Solution to frequency response                        */
/* to McGee's 31 - 1/3 rd octave filter bank.                     */
/* This algorithm has been written using papers from Dr. McGee. */
/* No particular attention was paid to the efficiency of this    */
/* algorithm.                                                     */
/*                                                                */
/* Code added to calculate group delay and phase delay and       */
/* differential time delay distortion.                            */
/*                                                                */
/* Same as transll.c.  But add phase unwrapping. This allows     */
/* observation of absolute delay and allows simpler coding of    */
/* group delay algorithm.                                         */
/*                                                                */
/*                                                                */
/*  Originally coded : November 27th, 1990.                       */
```

```
/*                                                              */
/*  Last modified    : December 5th, 1990.                      */
/*                                                              */
/****************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <carc.h>


#define MEM_LOC

#include <parades1.h>

#undef  MEM_LOC

#define N_OCT           11
#define PER_OCT         60
#define SAMP_FR      44100
#define P_THRES          3.0
#define N_THRES         -3.0

#define PAR_ONE         "parset1"   /* indicates use of parameter set #1
*/
#define PAR_TWO         "parset2"   /* indicates use of parameter set #2
*/
#define PAR_THREE "parset3"   /* indicates use of parameter set #3 */
#define PAR_FOUR  "parset4"   /* indicates use of parameter set #4 */
#define PAR_FIVE  "parset5"   /* indicates use of parameter set #5 */

/* declare stack size */
extern unsigned _stklen = 64000U;

/* functions protopype declarations */


void cmplx_mult(double ar,double ai,double br, double bi, double *mrr,
double *mri);
void c_div(double ar,double ai,double br, double bi, double *divr,
double *divi);
void parset1();

double vide[PER_OCT*N_OCT] = {0.0};


FILE *outputf;

int main (int argc, char *argv[])

{

FHEADER hdr;


/* pointers needed for input string comparisons   */
```

```
char *parset_1 = PAR_ONE;     /* indicates use of filter parset #1 */
char *parset_2 = PAR_TWO;     /* indicates use of filter parset #2 */
char *parset_3 = PAR_THREE;   /* indicates use of filter parset #3 */
char *parset_4 = PAR_FOUR;    /* indicates use of filter parset #4 */
char *parset_5 = PAR_FIVE;    /* indicates use of filter parset #5 */


static cmplxnum  ejw;      /* complex freq. variable at which    */
                     /* response is computed              */
static cmplxnum  sum;      /* temporary storage for summation    */
static cmplxnum  mult_res; /* complex multiplication result      */
static cmplxnum  div_res;  /* complex division result            */
static cmplxnum  num;
static cmplxnum  tden;              /* total denominator result storage
*/
static cmplxnum  H[PER_OCT*N_OCT]; /* complex transfer function result
*/
static cmplxnum  den_p[FILNUM];    /* temporary denominator result
storage */
                           /* for positive frequencies            */
static cmplxnum  den_n[FILNUM];    /* temporary denominator result
storage */
                           /* for negative frequencies            */

int i;
int j;
int k;
int l;
int freq_pts;                       /* total number of frequency points */

static double w[PER_OCT*N_OCT];    /* frequency pts at which response is
calculated */
                          /* in rads where 2*PI is sampling frequency
*/
                          /* points are log. spaced
*/
static double wfac;                /* relation between frequency points
*/
static double wff[PER_OCT*N_OCT]; /* frequency pts at which response is
calculated */
                          /* in Hz
*/
static double group_del[PER_OCT*N_OCT];    /* group delay response
*/
static double dif_time_del[PER_OCT*N_OCT];/* differential time delay
response      */
static double phase_del[PER_OCT*N_OCT];    /* group delay response
*/
static double mag[PER_OCT*N_OCT];      /* magnitude response
*/

static double phase[PER_OCT*N_OCT];        /* phase response in radians
*/
double wrap;                               /* number of times the phase
wrapped     */
                                 /* around 2*PI
*/
```

```
double new_phase[PER_OCT*N_OCT];          /* unwrapped phase response in
radians   */
char fname[32];
double rev_wff[PER_OCT*N_OCT];
double rev_H_re[PER_OCT*N_OCT];
double rev_H_im[PER_OCT*N_OCT];
double rev_mag[PER_OCT*N_OCT];
double rev_group_del[PER_OCT*N_OCT];
double rev_phase_del[PER_OCT*N_OCT];
double rev_new_phase[PER_OCT*N_OCT];
double rev_dif_time_del[PER_OCT*N_OCT];

freq_pts = PER_OCT*N_OCT;


InitHead(&hdr);

   CmdText (&hdr,argc,argv);
   if (strcmpi(argv[1],parset_1)==0)
   {
      printf ("The filter parameter set to be used is %s\n",PAR_ONE);
      parset1();
      AddText (&hdr," - Filter design #1, parameter set #1");
   }
   else if (strcmpi(argv[1],parset_2)==0)
   {
      printf ("The filter parameter set to be used is %s\n",PAR_TWO);
      parset2();
      AddText (&hdr," - Filter design #1, parameter set #2");
   }
   else if (strcmpi(argv[1],parset_3)==0)
   {
      printf ("The filter parset to be used is %s\n",PAR_THREE);
      parset3();
      AddText (&hdr," - Filter design #1, parameter set #3 - Real
weights");
   }
   else if (strcmpi(argv[1],parset_4)==0)
   {
      printf ("The filter parset to be used is %s\n",PAR_FOUR);
      parset4();
      AddText (&hdr," - Filter design #1, parameter set #4");
   }
   else if (strcmpi(argv[1],parset_5)==0)
   {
      printf ("The filter parset to be used is %s\n",PAR_FIVE);
      parset5();
      AddText (&hdr," - Filter design #1, parameter set #5");
   }
   else
   {
      printf ("\n\n*** Specified parameter set not recognized ***\n\n");
      printf ("Command format: trans20 parsetn \nwhere n is a number
from 1 to 9\n");
      exit(0);

   }
```

```
/* verify program so far */

   if ((outputf=fopen("trans20.txt","w")) == NULL)
   {
   printf("Error opening text file for writing\n");
   exit(0);
   }

/*    <------------start of commented out section

fprintf (outputf,"\t%s \t%s \t\t%s \t%s\n","Order","Freq","Wght","Wght
deg");

for (i=0;i<FILNUM;i++)
{
   fprintf (outputf,"\t%3d\t%lf\t%lf\t%lf\n",i+1,f[i],rr[i],wd[i]);

}

fprintf (outputf,"\t%s \t%s \t%s \n","Order","pp[i].re","pp[i].im");
for (i=0;i<FILNUM;i++)
{
   fprintf (outputf,"\t%3d\t%lf\t%lf\n",i+1,pp[i].re,pp[i].im);
}

fprintf (outputf,"\t%s \t%s \t%s \n","Order","pn[i].re","pn[i].im");
for (i=0;i<FILNUM;i++)
{
   fprintf (outputf,"\t%3d\t%lf\t%lf\n",i+1,pn[i].re,pn[i].im);
}

fprintf (outputf,"\t%s \t%s \t%s \n","Order","rr[i].re","rr[i].im");
for (i=0;i<FILNUM;i++)
{
   fprintf (outputf,"\t%3d\t%lf\t%lf\n",i+1,r[i].re,r[i].im);
}

fprintf (outputf,"\t%s \t%s \t%s \n","Order","wp[i].re","wp[i].im");
for (i=0;i<FILNUM;i++)
{
   fprintf (outputf,"\t%3d\t%lf\t%lf\n",i+1,wp[i].re,wp[i].im);
}

fprintf (outputf,"\t%s \t%s \t%s \n","Order","wn[i].re","wn[i].im");
for (i=0;i<FILNUM;i++)
{
   fprintf (outputf,"\t%3d\t%lf\t%lf\n",i+1,wn[i].re,wn[i].im);
}


end of commented out section ----------->   */
```

```
/* transfer function calculation                                     */
/* This program assumes that the sum of the r's equals 1.           */
/*                                                                   */
/*                    _                                   _          */
/*                   |  wp * r                 wn * r      |         */
/*         FILNUM -1 |    k   k                  k   k     |         */
/*            SUM    | ------------     +    ------------  |         */
/*            k=0    |  1 - pp *ejw           1 - pn *ejw  |         */
/*                   |_         k                    k    _|         */
/*  H(ejw) =   ------------------------------------------------      */
/*                    _                                   _          */
/*                   |      r                      r       |         */
/*         FILNUM -1 |       j                      j      |         */
/*            SUM    | ------------     +    ------------  |         */
/*            j=0    |  1 - pp *ejw           1 - pn *ejw  |         */
/*                   |_         j                    j    _|         */
/*                                                                   */
/*                                                                   */
/*********************************************************************/
/*                                                                   */
/*                                                                   */
/*   j is running index in denominator of transfer function         */
/*   k is running index in numerator   of transfer function         */
/*   i is running index of frequency points (ejw) does not          */
/*     appear in equation above                                      */
/*                                                                   */
/*                                              -jw                  */
/*  ejw is complex frequency variable e                              */
/*                                                                   */
/*                                                                   */
/*                                                                   */


w[0]=PI;
wfac = pow(2.0,1.0/PER_OCT);
wff[0] = SAMP_FR/2.0;
phase[0]=0;
new_phase[0]=0;
phase_del[0]=0;
group_del[0]=0;
dif_time_del[0]=0;

for (i=1;i<freq_pts;i++)
{
   w[i]=w[i-1]/wfac;
   wff[i]=(w[i]/(TWO_PI))*SAMP_FR;
   phase[i]=0;
   new_phase[i]=0;
   phase_del[i]=0;
   group_del[i]=0;
   dif_time_del[i]=0;

}

for (i=0;i<freq_pts;i++)
{
```

```
ejw.re=cos(w[i]);
ejw.im=-1.0*sin(w[i]);
sum.re=0.0;
sum.im=0.0;

/* transfer function denominator calculation */

for (j=0;j<FILNUM;j++)
{
   cmplx_mult(pp[j].re,pp[j].im,ejw.re,ejw.im, &mult_res.re,
   &mult_res.im);
   den_p[j].re = 1 - mult_res.re;
   den_p[j].im = -1.0 * mult_res.im;

   cmplx_mult(pn[j].re,pn[j].im,ejw.re,ejw.im, &mult_res.re,
   &mult_res.im);
   den_n[j].re = 1 - mult_res.re;
   den_n[j].im = -1.0 * mult_res.im;

   c_div(r[j].re,r[j].im,den_p[j].re,den_p[j].im, &div_res.re,
   &div_res.im);
   sum.re = sum.re + div_res.re;
   sum.im = sum.im + div_res.im;

   c_div(r[j].re,r[j].im,den_n[j].re,den_n[j].im, &div_res.re,
   &div_res.im);
   sum.re = sum.re + div_res.re;
   sum.im = sum.im + div_res.im;
}
tden.re = sum.re;
tden.im = sum.im;
sum.re = 0.0;
sum.im = 0.0;

/* transfer function numerator calculation */


for (k=0;k<FILNUM; k++)
{
   cmplx_mult(wp[k].re, wp[k].im, r[k].re, r[k].im, &mult_res.re,
   &mult_res.im);
   num.re = mult_res.re;
   num.im = mult_res.im;

   c_div(num.re, num.im, den_p[k].re, den_p[k].im, &div_res.re,
   &div_res.im);
   sum.re = sum.re + div_res.re;
   sum.im = sum.im + div_res.im;


   cmplx_mult(wn[k].re, wn[k].im, r[k].re, r[k].im, &mult_res.re,
   &mult_res.im);
   num.re = mult_res.re;
   num.im = mult_res.im;

   c_div(num.re, num.im, den_n[k].re, den_n[k].im, &div_res.re,
   &div_res.im);
```

```
      sum.re = sum.re + div_res.re;
      sum.im = sum.im + div_res.im;
   }

   /* total transfer function at w[i]     */

   c_div (sum.re, sum.im, tden.re, tden.im, &div_res.re, &div_res.im);


   /* store result */

   H[i].re = div_res.re;
   H[i].im = div_res.im;
}

/* print result to a file */


/*fprintf (outputf, "\t\t\%s\n","Filter Bank Transfer Function");
fprintf (outputf, "\t%s \t%s \t%s\n", "Freq(rad)","Real","Imaginary");

for (i=0;i<freq_pts-1;i++)
{
   fprintf (outputf, "\t%lf \t%lf \t%lf\n", w[i],H[i].re, H[i].im);
}
*/
fprintf (outputf, "\t%s \t%s \t%s \t%s\n", "Freq(rad)","Freq(Hz)",
"Magnitude","Phase(rad)");

for (i=0;i<freq_pts;i++)
{
   mag[i] = H[i].re * H[i].re + H[i].im * H[i].im;
   mag[i] = sqrt(mag[i]);
   phase[i] = atan2(H[i].im,H[i].re);

   fprintf (outputf, "\t%lf \t%lf \t%lf \t%lf\n", w[i],wff[i],
   mag[i], phase[i]);

}

/*  start of phase unwrapping algorithm       */

wrap=0;
new_phase[freq_pts-1]=phase[freq_pts-1];
for (i=freq_pts-2;i>=0;i--)
{
   if (phase[i+1]<0)
   {
      if ((phase[i+1] - phase[i]) < N_THRES)
      {
       wrap = wrap - TWO_PI;
      }
   }
   else
   {
      if ((phase[i+1] - phase[i]) > P_THRES)
      {
```

```
        wrap = wrap + TWO_PI;
       }

   }
   new_phase[i] = phase[i] + wrap;

}
/*    end of phase unwrapping algorithm       */



fprintf (outputf, "%s \t%s \t%s \t%s \t%s\n", "Freq(Hz)","Phase[i]",
"new_phase[i]","Group Delay(ms)");

for (i=1;i<freq_pts-1;i++)
{
   group_del[i]=-1.0*(new_phase[i-1]-new_phase[i+1])/((w[i-1]-
w[i+1])*SAMP_FR);

fprintf (outputf, "%lf \t%lf \t%lf \t%lf\n",wff[i],phase[i],
new_phase[i],1000*group_del[i]);
}


fprintf (outputf, "%s \t%s \t%s \t%s\n", "Freq(Hz)","Group Delay(ms)",
"Phase Delay(ms)","Diff. Time Delay Dist.(ms)");
for (i=freq_pts-1;i>=0;i--)
{
   phase_del[i]=-1.0*new_phase[i]/(wff[i]*TWO_PI);
   dif_time_del[i]=group_del[i] - phase_del[i];
   fprintf (outputf, "%lf \t%lf \t\t%lf
\t%lf\n",wff[i],1000*group_del[i],
   1000*phase_del[i],1000*dif_time_del[i]);
}

fclose (outputf);



/*  start of disabled section to write to text files */
/* writing the group delay on the text file */

/*
if ((outputf=fopen("group.dat","w")) == NULL)
{
   printf("Error opening group.dat text file for writing\n");
   exit(0);
}

for (i=freq_pts-3;i>=1;i--)
{
   fprintf (outputf,"%lf\n",group_del[i]);
}
fclose (outputf);
*/
```

```
/* writing the phase delay on the text file */
/*
if ((outputf=fopen("phasedel.dat","w")) == NULL)
{
   printf("Error opening phasedel.dat text file for writing\n");
   exit(0);
}

for (i=freq_pts-2;i>=1;i--)
{
   fprintf (outputf,"%lf\n",phase_del[i]);
}
fclose (outputf);
*/

/* writing the differential time delay distortion on the text file */
/*
if ((outputf=fopen("diftime.dat","w")) == NULL)
{
   printf("Error opening diftime.dat text file for writing\n");
   exit(0);
}

for (i=freq_pts-3;i>=1;i--)
{
   fprintf (outputf,"%lf\n",dif_time_del[i]);
}
fclose (outputf);
*/

/*  end of disabled section to write to text files */


/*  reversing order of frequency points - ascending order instead of */
/* descending order for CARC output.                                 */


for (i=0;i<freq_pts;i++)
{
   rev_wff[i]=wff[freq_pts-1-i];
   rev_H_re[i]=H[freq_pts-1-i].re;
   rev_H_im[i]=H[freq_pts-1-i].im;
   rev_mag[i]=mag[freq_pts-1-i];
   rev_group_del[i]=group_del[freq_pts-1-i];
   rev_new_phase[i]=new_phase[freq_pts-1-i];
   rev_phase_del[i]=phase_del[freq_pts-1-i];
   rev_dif_time_del[i]=dif_time_del[freq_pts-1-i];

}

hdr.TimeFreq = D_FREQ;  /* frequency domain data */

hdr.Imag  = (void *) vide;
hdr.NData =PER_OCT*N_OCT;
hdr.Precision = P_REAL64;
hdr.Format    = F_INDEXED;
```

```
/* writing the group delay on the carc file */
   strcpy(fname,"group.car");
   hdr.Index = &rev_wff;
   hdr.Real  = &rev_group_del;

   i = WritFile( &hdr, fname );
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }


/* writing the phase delay on the carc file */
   strcpy(fname,"phasedel.car");
   hdr.Index = &rev_wff;
   hdr.Real  = &rev_phase_del;

   i = WritFile( &hdr, fname );
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }


/* writing the differential time delay distortion on the carc file */
   strcpy(fname,"diftime.car");
   hdr.Index = &rev_wff;
   hdr.Real  = &rev_dif_time_del;

   i = WritFile( &hdr, fname );
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }

/* writing the unwrapped phase response of the filter bank on the carc
file */
   strcpy(fname,"phase.car");
   hdr.Index = &rev_wff;
   hdr.Real  = &rev_new_phase;

   i = WritFile( &hdr, fname );
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }


/* writing the frequency response of the filter bank on the carc file */
   strcpy(fname,"h.car");
   hdr.Index = &rev_wff;
   hdr.Real  = &rev_H_re;
   hdr.Imag  = &rev_H_im;

   i = WritFile( &hdr, fname );
```

```
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }


/* writing the magnitude response of the filter bank on the carc file */
   strcpy(fname,"mag.car");
   hdr.Index = &rev_wff;
   hdr.Real  = &rev_mag;
   hdr.Imag  = (void *) vide;

   i = WritFile( &hdr, fname );
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }


   exit(0);
   return(0);
}

void cmplx_mult (double ar, double ai, double br, double bi, double
*mrr,
double *mri)
{
    *mrr = ar * br - ai * bi;
    *mri = bi * ar + ai * br;
}
void c_div (double ar, double ai, double br, double bi, double *divr,
double *divi)
{
   double den;

   den = br*br + bi*bi;
   *divr = (ar*br + ai*bi)/den;
   *divi = (ai*br - ar*bi)/den;
}
```

```
/* paradesl.h */
/* header file for desl of 31 - 1/3 octave filter bank */

#define FILNUM          31
#define RFAC          1553.445297
#define THIRD_OCT        1.25992105
#define PI               3.141592654
#define TWO_PI           6.283185308

typedef struct {
      double re;
      double im;
      } cmplxnum;

MEM_LOC double f[FILNUM]; /* pole center frequencies in degrees where
360        */
                        /* corresponds to the sampling frequency     */
MEM_LOC double rr[FILNUM];    /* forward "Kalman" gain magnitude    */
MEM_LOC double wd[FILNUM];    /* angle in degree of every output weight
*/

MEM_LOC cmplxnum  yp[FILNUM]; /* intermediate result storage for
positive frequencies*/
MEM_LOC cmplxnum  yn[FILNUM]; /* intermediate result storage for
negative frequencies*/
MEM_LOC cmplxnum  pp[FILNUM]; /* complex positive center frequencies
*/
MEM_LOC cmplxnum  pn[FILNUM]; /* complex negative center frequencies
*/
MEM_LOC cmplxnum  r[FILNUM];  /* complex forward Kalman gains       */
MEM_LOC cmplxnum  wp[FILNUM]; /* complex output weights for positive
frequencies     */
MEM_LOC cmplxnum  wn[FILNUM]; /* complex output weights for negative
frequencies     */
```

```
void parset1()
{

#define MEM_LOC   extern  /* declare variables as external */
                          /* in header file parades1.h     */
#include <parades1.h>
#undef  MEM_LOC

#include <math.h>

int k; /* running filter index   */
double arg;
/* initialize McGee's constants */

/* initialize filter bank */

yp[0].re=0.0;
yp[0].im=0.0;
yn[0].re=0.0;
yn[0].im=0.0;
f[0]=160.36177;
rr[0]=f[0]/RFAC;
r[0].im=0.0;


for (k=1;k<FILNUM;k++)
{

/* initialize filter registers */
   yp[k].re=0.0;
   yp[k].im=0.0;
   yn[k].re=0.0;
   yn[k].im=0.0;
   r[k].im=0.0;
   f[k]=f[k-1]/THIRD_OCT;
   rr[k]=f[k]/RFAC;
}
wd[0]=104.98900;
wd[1]=-61.66708;
wd[2]= 99.94393;
wd[3]=-82.03426;
wd[4]= 77.68593;
wd[5]=-104.52081;
wd[6]=  55.91988;
wd[7]=-124.64165;
wd[8]=  37.02428;
wd[9]=-140.65688;
wd[10]= 22.25593;
wd[11]=-151.89099;
wd[12]=  11.51269;
wd[13]=-158.92077;
wd[14]=   3.79450;
wd[15]=-162.76066;
wd[16]=  -2.00559;
wd[17]=-164.27886;
wd[18]=  -6.78053;
```

```
wd[19]=-164.05669;
wd[20]= -11.19704;
wd[21]=-162.42334;
wd[22]= -15.77170;
wd[23]=-159.51276;
wd[24]= -20.96613;
wd[25]=-155.28792;
wd[26]= -27.33998;
wd[27]=-149.47748;
wd[28]= -36.06589;
wd[29]=-141.20338;
wd[30]= -53.73248;

/* initialize filter coefficients and weights using McGee's values */


for (k=0;k<FILNUM;k++)
{

   pp[k].re=cos(TWO_PI*f[k]/360.0);
   pp[k].im=sin(TWO_PI*f[k]/360.0);
   pn[k].re=pp[k].re;
   pn[k].im=pp[k].im * -1.0;
   r[k].re=rr[k];

   arg=TWO_PI*wd[k]/360.0;
   wp[k].re=cos(arg);

   wp[k].im=sin(TWO_PI*wd[k]/360.0);
   wn[k].re=wp[k].re;
   wn[k].im=wp[k].im*-1.0;
}


}
```

Appendix B - Filterbank Floating Point Simulation

```
/***************************************************************/
/*   file = simacl2.c                                          */
/*                                                             */
/*   at DOS prompt: simacl2 infile outfile                     */
/*                                                             */
/*                                                             */
/*                                                             */
/* This file was copied from simacl1.c .                       */
/* same as simacl1.c except that input and output file names*/
/* are taken from the command line.  Also SKIP and counter  */
/* code were deleted for faster execution.                     */
/***************************************************************/

/* McGee's filter bank : 31 1/3 rd octave filters             */
/* floating point simulation program                           */
/* Takes an input file with 16-bit samples and processes it */
/* implementing a 31 - 1/3rd octave filter bank.               */
/* The filter bank parameters were obtained from professor  */
/* McGee at the University of Ottawa.                          */


/*                                                  */
/*  Algorithm coded by Marc E. Bonneville           */
/*                                                  */
/*     Nov. 11, 1990.                               */

#include <time.h>
#include <dos.h>
#include <math.h>
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys\stat.h>


#define END_OF_FILE 0
#define CONTINUE 1
#define NUM_BYTES 2

#define FILNUM          31
#define PI         3.141592654
#define RFAC            1553.445297
#define THIRD_OCT       1.25992105
#define double          float

int main (int argc, char *argv[])
{
time_t first,second;


static double  x_re,yp_re[FILNUM], yn_re[FILNUM], pp_re[FILNUM],
pn_re[FILNUM];
```

```
static double  x_im,yp_im[FILNUM], yn_im[FILNUM], pp_im[FILNUM],
pn_im[FILNUM];
static double  r_re[FILNUM], w_re[FILNUM],
wn_re[FILNUM],t_mult_re[FILNUM];
static double  r_im[FILNUM], w_im[FILNUM], wn_im[FILNUM];
static double  e_re, out_re, sum_re, temp_re, mult_res_re;
static double  e_im, out_im, sum_im, temp_im, mult_res_im;
double maxout,minout;
register int i;
double f[FILNUM], rr[FILNUM], wd[FILNUM];
int rd_handle, wr_handle, rd_sample, wr_sample, rd_status, wr_status;
int *rd_buf,*wr_buf;
/* initialize McGee's constants */

/* initialize filter bank */

maxout=0.0;
minout=0.0;
x_re = 0.0;
x_im = 0.0;
yp_re[0]=0.0;
yp_im[0]=0.0;
yn_re[0]=0.0;
yn_im[0]=0.0;
f[0]=160.36177;
rr[0]=f[0]/RFAC;
r_im[0]=0.0;

for (i=1;i<FILNUM;i++)
{

/* initialize filter registers */
   yp_re[i]=0.0;
   yp_im[i]=0.0;
   yn_re[i]=0.0;
   yn_im[i]=0.0;
   r_im[i]=0.0;
   f[i]=f[i-1]/THIRD_OCT;
   rr[i]=f[i]/RFAC;
}
wd[0]=104.98900;
wd[1]=-61.66708;
wd[2]= 99.94393;
wd[3]=-82.03426;
wd[4]= 77.68593;
wd[5]=-104.52081;
wd[6]=  55.91988;
wd[7]=-124.64165;
wd[8]=  37.02428;
wd[9]=-140.65688;
wd[10]= 22.25593;
wd[11]=-151.89099;
wd[12]=  11.51269;
wd[13]=-158.92077;
wd[14]=   3.79450;
wd[15]=-162.76066;
wd[16]=  -2.00559;
```

```
wd[17]=-164.27886;
wd[18]=  -6.78053;
wd[19]=-164.05669;
wd[20]= -11.19704;
wd[21]=-162.42334;
wd[22]= -15.77170;
wd[23]=-159.51276;
wd[24]= -20.96613;
wd[25]=-155.28792;
wd[26]= -27.33998;
wd[27]=-149.47748;
wd[28]= -36.06589;
wd[29]=-141.20338;
wd[30]= -53.73248;

/* initialize filter coefficients and weights using McGee's values */


for (i=0;i<FILNUM;i++)
{
   pp_re[i]=cos(2.0*PI*f[i]/360.0);
   pp_im[i]=sin(2.0*PI*f[i]/360.0);
   pn_re[i]=pp_re[i];
   pn_im[i]=pp_im[i] * -1.0;
   r_re[i]=rr[i];
   w_re[i]=cos(2.0*PI*wd[i]/360.0);
   w_im[i]=sin(2.0*PI*wd[i]/360.0);
   wn_re[i]=w_re[i];
   wn_im[i]=w_im[i]*-1.0;
}


/* filter code, in output transfer */
   if ((rd_handle= open(argv[1],  O_RDONLY | O_BINARY)) == -1)
   {
      printf ("Error Opening Read File\n");
      exit(1);
   }
   if ((wr_handle= open(argv[2],   O_RDWR | O_CREAT | O_TRUNC |
   O_BINARY,S_IWRITE)) == -1)
   {
      printf ("Error Opening Write File\n");
      exit(1);
   }

   printf("\n");
   rd_status = CONTINUE;


   first=time(NULL);


   /* Start of outer loop                                        */
   /* Read and process input file until end-of-file is reached   */


   while (rd_status!=END_OF_FILE)
```

```
{
  if ((rd_status = read(rd_handle,rd_buf,NUM_BYTES)) == -1)
  {
   printf ("Read Failed.\n");
   exit(1);
  }
  else
  {
   if (rd_status==0)
   {
      rd_status=END_OF_FILE;
   }
   else
   {
         rd_sample=*rd_buf;

         x_re = _rotl(rd_sample,8);  /* swap upper and lower bytes
*/
                           /* for Mac compatibility        */
         sum_re=0.0;

         /* error calculation */

         /*                FILNUM-1                                  */
         /* e(t) = x(t) - SUM     yp (t-1) * pp  + yn (t-1) * pn  */
         /*                 i=0      i            i     i          i */
         /*                                                          */


         for (i=0;i<FILNUM;i++)
         {

          t_mult_re[i] = yp_re[i] * pp_re[i] - yp_im[i] * pp_im[i];
          sum_re += t_mult_re[i];

         }
         e_re = x_re - 2.0*sum_re;


         /*     output calculation                                  */
         /*                                                         */
         /*          FILNUM-1                                       */
         /*  out(t) = SUM    [ w [ r *e(t) + pp *yp (t-1) ]        */
         /*           i=0      i   i          i   i                 */
         /*                                                         */
         /*                  + wn [ r *e(t) + pn *yn (t-1) ] ]     */
         /*                      i   i         i   i                */

         /*                                            */
         /*  intermediate result calculation           */
         /*                                            */
         /*  yp (t) = r *e(t) + pp *yp (t-1)           */
         /*    i       i         i   i                 */
         /*                                            */
         /*  yn (t) = r *e(t) + pn *yn (t-1)           */
```

```
            /*    i        i          i   i                */
            /*                                             */

            sum_re = 0.0;

            for (i=0;i<FILNUM;i++)
            {

             yp_im[i] = pp_re[i] * yp_im[i] + pp_im[i] * yp_re[i];
             yp_re[i] = t_mult_re[i] + r_re[i] * e_re;

             sum_re +=(yp_re[i]) * w_re[i] - (yp_im[i]) * w_im[i];

            }

            out_re=2.0*sum_re;


            /* Convert floating point to 16-bit integer            */
            /* and swap upper and lower bytes for mac compatibility */

            wr_sample=out_re;
            wr_sample = _rotl(wr_sample,8);

            /* Determine largest and smallest floating point output  */
            /* Print result on screen to verify that maximum output  */
            /* sample does not exceed 32,767 and that minimum output */
            /* is not smaller than -32,768                           */

            if (out_re>maxout)
             maxout=out_re;
            if (out_re<minout)
             minout=out_re;

            wr_buf=&wr_sample;
            if ((wr_status = write(wr_handle, wr_buf,NUM_BYTES)) == -1)
            {
             printf ("Write Failed.\n");
             exit(1);
            }

      }
     }
   }
second=time(NULL);

close(rd_handle);
close(wr_handle);

printf ("Input  binary file  = %s\n",argv[1]);
printf ("Output binary file  = %s\n",argv[2]);
printf ("Maximum output signal = %lf\n",maxout);
printf ("Minimum output signal = %lf\n",minout);
printf ("Run time was : %f seconds\n\n",difftime(second,first));

return 0;
```

}

Appendix C - Empirical Optimum Scaling Factor

```
/*******************************************************************/
/*   file opt2.c:     calculates the mismatch error  when          */
/*                    a generalized 12.041 dB boost curve           */
/*                    is scaled to another generalized curve.       */
/*                    The allows an empirical determination of      */
/*                    the optimum gain scaling method for a         */
/*                    second order bandpass filter.                 */
/*
/*                    Copied from opt1.c.
/*
/*                                                                  */
/*                                                                  */
/*                                                                  */
/*  Originally coded : April 1st, 1991.                             */
/*                                                                  */
/*  Last modified    : April 1st, 1991.                             */
/*******************************************************************/


#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <carc.h>


/* define constants    */

#define NUM_OCT         11          /* number of octave covered by
measurement  */
#define THIRD_OCT        1.25992105 /* ratio of frequencies defining one
third octave */
#define PTS_PER_OCT      5          /* number of points per octave for
measurement  */
#define PI               3.141592654
#define SAMP_FR       44100         /* sampling frequency   */
#define TWO_PI           6.283185308

#define MAX_BOOST       12.041
#define NUM_aa          50              /* number of aa values to
calculate */
#define START_aa        0.1         /* starting value of aa */
#define END_aa          1.5         /* ending value of aa */
#define NUM_b           100             /* number of b values to
calculate */
#define START_b         -0.8        /* starting value of b */
#define END_b           -0.05       /* ending value of b */
#define NUM_A            3              /* number of A values to
calculate */
#define START_A          0.6        /* starting value of A */
#define END_A            1.0        /* ending value of A */
```

```
#define BIQ_Q           1           /* biquad filter quality factor */


/* declare stack size max. */
extern unsigned _stklen=64000U;

/* functions protopype declarations */

double vide[PTS_PER_OCT*NUM_OCT] = {0.0};


FILE *outputf;

int main ()

{
FHEADER hdr;


typedef struct {
      double re;
      double im;
      } cmplxnum;


static double  wo;           /* digital center frequency in radians */
static double  fo;           /* center frequency in Hz */
static double  measw[PTS_PER_OCT*NUM_OCT];  /* freq. in rads at which
measurement is taken */
static double  measf[PTS_PER_OCT*NUM_OCT];  /* freq. in Hz at which
measurement is taken */

static double  max_norm_error_all_theta;
static double  arg1;         /* temporary storage variable to ease
coding */
static double  arg2;         /* temporary storage variable to ease
coding */
static double  arg_max;      /* temporary storage variable to ease
coding */
static double  theta;        /* temporary storage variable to ease
coding */
static double  theta_sq;           /* temporary storage variable to
ease coding */
static double  Q;      /* filter quality factor                   */
static double freq_pts; /* total number of measuring frequencies     */

static double error;     /* mismatch error between scaled and actual  */
static double norm_error;/* normalized mismatch error between scaled and
actual  */
static double aa[NUM_aa];      /* use aa instead of a to avoid confusion
with A - refer to March 26 */
static double b[NUM_b];  /* denominator factor refer to March 26  */
static double A[NUM_A];  /* gain factor in generalized 12.041 dB curve
*/
static double best_max_norm_error[NUM_A][NUM_aa]; /* max. normalized
mismatch error  */
```

```
static double best_b[NUM_A][NUM_aa]; /* value of b at which max.
normalized mismatch error occurred  */
static double step_aa;   /* step size between aa values */
static double step_b;   /* step size between b values */
static double step_A;   /* step size between A values */
static double x;        /* ideal ratio of max. boost curve to new curve
*/

int i_f;   /* frequency running index */
int i_aa;  /* aa running index */
int i_b;   /* b running index */
int i_A;   /* A running index */
int i;   /*  */

double wfac; /* ratio of consecutive frequencies at which measurements
are taken */
char fname[32];

InitHead(&hdr);

freq_pts = PTS_PER_OCT*NUM_OCT;

/* define filter parameters  */
/* fo = 7500.0; */

/*fo = 8306;  */
fo = 750;

Q=BIQ_Q;
wo = fo*TWO_PI;  /*  center frequency in rads/s */

/* define values for aa and b */

step_aa=(END_aa - START_aa)/(NUM_aa-1);
step_b=(END_b - START_b)/(NUM_b-1);
step_A=(END_A - START_A)/(NUM_A-1);

if ((outputf=fopen("opt2aab.txt","w")) == NULL)
{
   printf("Error opening text file for writing\n");
   exit(0);
}

aa[0]=START_aa;
fprintf (outputf,"\t%s %lf\n\n","aa[0]=",aa[0]);

for (i_aa=1;i_aa<NUM_aa;i_aa++)
{
   aa[i_aa]=aa[i_aa-1] + step_aa;
   if (aa[i_aa]==-1)
   {
       aa[i_aa]=-0.9;
   }
   fprintf (outputf,"\t%s %d %s %lf\n\n","aa[",i_aa,"]=",aa[i_aa]);
}

b[0]=START_b;
```

```
fprintf (outputf,"\t%s %lf\n\n","b[0]=",b[0]);

for (i_b=1;i_b<NUM_b;i_b++)
{
   b[i_b]=b[i_b-1] + step_b;
   if (b[i_b]==0)
   {
       b[i_b]=0.1;
   }
   if (b[i_b]==-1)
   {
       b[i_b]=-0.9;
   }
   fprintf (outputf,"\t%s %d %s %lf\n\n","b[",i_b,"]=",b[i_b]);
}

A[0]=START_A;
fprintf (outputf,"\t%s %lf\n\n","A[0]=",A[0]);

for (i_A=1;i_A<NUM_A;i_A++)
{
   A[i_A]=A[i_A-1] + step_A;
   if (A[i_A]==0)
   {
       A[i_A]=0.1;
   }
   fprintf (outputf,"\t%s %d %s %lf\n\n","A[",i_A,"]=",A[i_A]);
}

fclose(outputf);


if ((outputf=fopen("opt2.txt","w")) == NULL)
{
   printf("Error opening text file for writing\n");
   exit(0);
}


wfac = pow(2.0,1.0/PTS_PER_OCT);
measw[freq_pts-1]=PI;
measf[freq_pts-1]=measw[freq_pts-1]*SAMP_FR/TWO_PI;


for (i_f=freq_pts-2;i_f>=0;i_f--)
{
   measw[i_f]=measw[i_f+1]/wfac;
   measf[i_f]=measf[i_f+1]/wfac;
}

norm_error=0;
for (i_A=0;i_A<NUM_A;i_A++)
{
   for (i_aa=0;i_aa<NUM_aa;i_aa++)
   {
      best_max_norm_error[i_A][i_aa]=1000.0;
   }
```

```
}

for (i_A=0;i_A<NUM_A;i_A++)
{
   printf("\t%s %d %s %lf\n","A[",i_A,"]=",A[i_A]);
   fprintf (outputf,"\n\n\n\t%s %lf\n\n","A[i_A]=",A[i_A]);
   fprintf (outputf,"\t%s \t\t%s \n","aa","best_max_norm_error");

   for (i_aa=0;i_aa<NUM_aa;i_aa++)
   {
/*      printf("\t%s %d \n","i_aa=",i_aa);*/
      for (i_b=0;i_b<NUM_b;i_b++)
      {
       if (aa[i_aa]!=b[i_b])
       {

      /*      printf("\t%s %d \n","i_b=",i_b);*/

          if (b[i_b]==-1)
          {
             printf("\t%s %lf\n","b[]=",b[i_b]);
          }

          /* calculate boost at center frequency (theta=0) */
          arg_max=(1+aa[i_aa])/(1+b[i_b]);
          if (arg_max<0)
          {
             arg_max=arg_max*-1.0;
          }

          if (arg_max<=0)
          {
             printf("\t%s %lf\n","arg_max=",arg_max);
          }


          x=20*log10(arg_max)/MAX_BOOST;
/*        printf("\t%s \n","x calculation has been done.");*/

          max_norm_error_all_theta=0.0;
          /* find max. error for all values of theta (ie. frequencies)
*/

          for (i_f=0;i_f<freq_pts;i_f++)
          {

             theta=Q*(fo/measf[i_f] - measf[i_f]/fo);
             theta_sq=theta*theta;

arg1=((1+aa[i_aa])*(1+aa[i_aa])+theta_sq)/((1+b[i_b])*(1+b[i_b])+theta_s
q);

arg2=((1+A[i_A])*(1+A[i_A])+theta_sq)/((1+A[i_A])*(1+A[i_A])/16 +
theta_sq);
/*           printf("\t%s \n","before error calculation.");*/
/*           printf("\t%s %lf\n","arg1=",arg1);*/
/*           printf("\t%s %lf\n","arg2=",arg2);*/
```

```
/*              printf("\t%s %lf\n","x=",x);*/


      /* instead of taking square root of arg1 and arg2, use 10*log*/
                error=10.0*log10(arg1) - x*10.0*log10(arg2);
/*              printf("\t%s %lf\n","after error calculation.
arg_max=",arg_max);*/
                if (error==0.0)
                {
                 norm_error=0.0;
                }
                else
                {
                 norm_error=error/(20*log10(arg_max));
                }
/*              printf("\t%s \n","after norm_error calculation.");*/
                if (fabs(norm_error)>fabs(max_norm_error_all_theta))
                {
/*               printf("\t%s \n","max error check");*/
                 max_norm_error_all_theta=norm_error; /* max. normalized
error for all frequencies */
                }
            }

            if
(fabs(max_norm_error_all_theta)<fabs(best_max_norm_error[i_A][i_aa]))
            {

best_max_norm_error[i_A][i_aa]=fabs(max_norm_error_all_theta);
                best_b[i_A][i_aa]=b[i_b];
            }

        /*   printf("\t%s \n","after frequency sweep");*/
        }
      }
      fprintf (outputf,"\t%lf
\t%lf\n",aa[i_aa],best_max_norm_error[i_A][i_aa]);
   }
}

/* print best_b as a function of aa */

printf("\t%s \n","printing best_b vs aa to text file.");

for (i_A=0;i_A<NUM_A;i_A++)
{
   fprintf (outputf,"\t%s %lf\n\n","A=",A[i_A]);
   fprintf (outputf,"\t%s \t\t%s \n","aa","best_b");
   for (i_aa=0;i_aa<NUM_aa;i_aa++)
   {
      fprintf (outputf,"\t%lf \t%lf\n",aa[i_aa],best_b[i_A][i_aa]);
   }
}


   fclose(outputf);
```

```
   AddText (&hdr,"Empirical Optimum Error Curves ");
   AddText (&hdr,"Center frequency = 750Hz, Q=1, \n");
   AddText (&hdr,"OPT2.C, M. E. Bonneville, April 1st, 1991.");

   hdr.TimeFreq = D_TIME;  /* time domain data although fictitious ie.
aa is time*/

   hdr.NData =NUM_aa;
   hdr.Precision = P_REAL64;
   hdr.Format    = F_INDEXED;

/* writing transfer function on the carc file */
   strcpy(fname,"opte06.car");
   hdr.Index = &aa[0];
   hdr.Real  = &best_max_norm_error[0][0];
   hdr.Imag  = (void *) vide;

   i = WritFile( &hdr, fname );
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }
/* writing transfer function on the carc file */
   strcpy(fname,"optb06.car");
   hdr.Index = &aa[0];
   hdr.Real  = &best_b[0][0];
   hdr.Imag  = (void *) vide;

   i = WritFile( &hdr, fname );
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }
/* writing transfer function on the carc file */
   strcpy(fname,"opte08.car");
   hdr.Index = &aa[0];
   hdr.Real  = &best_max_norm_error[1][0];
   hdr.Imag  = (void *) vide;

   i = WritFile( &hdr, fname );
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }
/* writing transfer function on the carc file */
   strcpy(fname,"optb08.car");
   hdr.Index = &aa[0];
   hdr.Real  = &best_b[1][0];
   hdr.Imag  = (void *) vide;

   i = WritFile( &hdr, fname );
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }
/* writing transfer function on the carc file */
   strcpy(fname,"opte10.car");
   hdr.Index = &aa[0];
   hdr.Real  = &best_max_norm_error[2][0];
   hdr.Imag  = (void *) vide;
```

```
   i = WritFile( &hdr, fname );
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }
/* writing transfer function on the carc file */
   strcpy(fname,"optb10.car");
   hdr.Index = &aa[0];
   hdr.Real  = &best_b[2][0];
   hdr.Imag  = (void *) vide;

   i = WritFile( &hdr, fname );
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }

   exit(0);
   return(0);
}
```

Appendix D - Automation Algorithms

Biquadratic Filter

```
/***********************************************************************/
/*  file adapsec3.c  :  reads in two frequency indexed CARC files,     */
/*                  one target and one measurement.                     */
/*                  Then this program uses the functions in adapt       */
/*                  to optimizive in succession N_STAGES of             */
/*                  second order biquads.  Then the frequency           */
/*                  response of each stage is written to a CARC         */
/*                  file for subsequent examination.  In addition       */
/*                  a CARC file is written with the resulting           */
/*                  frequency response.                                  */
/*                                                                       */
/*              format : ADAPSEC3 target_file measurement_file           */
/*                                                                       */
/*                                                                       */
/*  Originally coded : June 19th, 1991.                                  */
/*                                                                       */
/*  Last modified    : June 19th, 1991.                                  */
/*                                                                       */
/***********************************************************************/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <carc.h>

/* define constants    */
#define exp             2.718281828
#define DEFAULT_SSE     1000000.0
#define N_STAGES  15
#define MAX_N_STAGES    20


/* declare stack size max. */
extern unsigned _stklen=32000U;

/* function prototype declaration */
REAL64 ssel(REAL64 *targ, REAL64 *resp, INT16 ndata);

INT16 centr_fl(INT16 ndata, REAL64 *f, REAL64 *m,
             REAL64 *cent_fo, REAL64 *der);

INT16 peakl(INT16 ndata, REAL64 *f, REAL64 *m, REAL64 *cent_fo,
             REAL64 *der);

INT16 dipl(INT16 ndata, REAL64 *f, REAL64 *m, REAL64 *cent_fo,
             REAL64 *der);
```

```
REAL64 **matrixd(int nrows, int ncols);
REAL64 *vectord(int nrows);
int    *vectori(int nrows);

INT16 approx_s (REAL64 K, REAL64 fo, REAL64 Q, INT16 ndata,
             REAL64 *f, REAL64 *resp_dB);

INT16 exact_s (REAL64 K, REAL64 fo, REAL64 Q, INT16 ndata,
             REAL64 *f, REAL64 *resp_dB);



INT16 opt_sec3(INT16 ndata1, REAL64 *f, REAL64 *m, REAL64 fo, REAL64
*best_Q,
              REAL64 *best_alpha);



INT16 rect2(INT16 ndata, REAL64 *f, REAL64 *m, REAL64 *work_m, REAL64
fo);

REAL64 norm_max(INT16 ndata1, REAL64 *resp_dB);
REAL64 norm_min(INT16 ndata1, REAL64 *resp_dB);

/*------------------------------------*/

FILE *outputf;
int main (int argc, char *argv[])

{
   FHEADER hdr1; /* input data structure 1 = target file */
   FHEADER hdr2; /* input data structure 2 = measurement file */

   INT16    ndata;
   INT16    i;
   INT16    j;
   INT16    N_peaks_AND_dips;
   INT16    dummy_int;


   REAL64   gain;
   REAL64   n_min;
   REAL64   n_max;
   REAL64   *m;
   REAL64   *tar_m;
   REAL64   *work_m;
   REAL64   *target;
   REAL64   *m_max;
   REAL64   *m_min;
   REAL64   *flat;
   REAL64   *f;
   REAL64   *peaks_AND_dips;
   REAL64   *der;
   REAL64   *app_sec;
   REAL64   *eq_stage_out;
   REAL64   *exa_sec;
```

```
REAL64  sse_min;
REAL64  sse_max;
REAL64  Q;
REAL64  alpha;
REAL64  K;
REAL64  best_Q;
REAL64  best_alpha;
REAL64  best_sse;
REAL64  best_fo;
REAL64  sse;

typedef struct {
      double K;
      double Q;
      double fo;
      } stage_id;

stage_id stage_info[N_STAGES];
static char *stage_file[MAX_N_STAGES] = {"stage1.car",
                                "stage2.car",
                                "stage3.car",
                                "stage4.car",
                                "stage5.car",
                                "stage6.car",
                                "stage7.car",
                                "stage8.car",
                                "stage9.car",
                                "stage10.car",
                                "stage11.car",
                                "stage12.car",
                                "stage13.car",
                                "stage14.car",
                                "stage15.car",
                                "stage16.car",
                                "stage17.car",
                                "stage18.car",
                                "stage19.car",
                                "stage20.car"};

static char *exact_file[MAX_N_STAGES] = {"exact1.car",
                                "exact2.car",
                                "exact3.car",
                                "exact4.car",
                                "exact5.car",
                                "exact6.car",
                                "exact7.car",
                                "exact8.car",
                                "exact9.car",
                                "exact10.car",
                                "exact11.car",
                                "exact12.car",
                                "exact13.car",
                                "exact14.car",
                                "exact15.car",
                                "exact16.car",
                                "exact17.car",
                                "exact18.car",
```

```
                                    "exact19.car",
                                    "exact20.car"};

    static char *stage_out[MAX_N_STAGES] = {"secout1.car",
                                    "secout2.car",
                                    "secout3.car",
                                    "secout4.car",
                                    "secout5.car",
                                    "secout6.car",
                                    "secout7.car",
                                    "secout8.car",
                                    "secout9.car",
                                    "secout10.car",
                                    "secout11.car",
                                    "secout12.car",
                                    "secout13.car",
                                    "secout14.car",
                                    "secout15.car",
                                    "secout16.car",
                                    "secout17.car",
                                    "secout18.car",
                                    "secout19.car",
                                    "secout20.car"};

    InitHead(&hdr1); /* initialize structure */
    InitHead(&hdr2); /* initialize structure */

    i = ReadFile( &hdr1, argv[1]);
    if ( i!= 0 )
    {
       PrtErr(FCT_READFILE,i);
    }

    if (hdr1.TimeFreq != D_FREQ )
    {
       printf("Input file1 is not in the frequency domain.\n");
       exit(0);
    }

    if (hdr1.Format != F_INDEXED )
    {
       printf("Input file1 format is not frequency indexed.\n");
       exit(0);
    }

    /* read second file */

    i = ReadFile( &hdr2, argv[2]);
    if ( i!= 0 )
    {
       PrtErr(FCT_READFILE,i);
    }

    if (hdr2.TimeFreq != D_FREQ )
    {
       printf("Input file2 is not in the frequency domain.\n");
       exit(0);
```

```
}

if (hdr2.Format != F_INDEXED )
{
   printf("Input file2 format is not frequency indexed.\n");
   exit(0);
}

ndata = hdr1.NData;

if (hdr1.NData != hdr2.NData )
{
   printf("Input files not same length.\n");
   exit(0);
}

/*--------------------------------------------------*/
/* memory allocation                              */

work_m          = vectord(ndata);
tar_m           = vectord(ndata);
eq_stage_out    = vectord(ndata);
m_min           = vectord(ndata);
m_max           = vectord(ndata);
flat            = vectord(ndata);
peaks_AND_dips  = vectord(ndata);
app_sec         = vectord(ndata);
exa_sec         = vectord(ndata);
der             = vectord(ndata);


/*--------------------------------------------------*/
/* initialize variables                           */

m      = (REAL64 *) hdr2.Real;
target = (REAL64 *) hdr1.Real;
f      = (REAL64 *) hdr2.Index;
gain = 0.0;


/*--------------------------------------------------*/
/* subtract target and initialize flat            */

for (i=0; i<ndata; i++)
{
   printf ("\nm[%d] = %lf",i, m[i]);
   tar_m[i]    = target[i] - m[i];
   flat[i] = 0.0;
}

/* open file for log data */

   if ((outputf=fopen("adapsec3.log","w")) == NULL)
   {
    printf("\n\nError opening adapsec3.log for writing.\n");
    exit(0);
   }
```

```
/*-----------------------------------------------------*/
/* Find best least squares approximation one stage */
/* at a time.                                       */

for (j=0; j<N_STAGES; j++)
{
   best_Q = 0.0;
   best_fo = 0.0;
   best_alpha = 0.0;
   best_sse = DEFAULT_SSE;


   /*-----------------------------------------------------------*/

   printf ("\nRunning function centr_f1\n");
   N_peaks_AND_dips = centr_f1(ndata, f, tar_m,
                        peaks_AND_dips, der);
   printf ("\nReturned from centr_f1 \nN_peaks_AND_dips =
%d",N_peaks_AND_dips);

   printf ("\nOptimizing stage # %d", j+1);


   /* find best biquad for every peak or dip and keep best */
   for (i=0; i<N_peaks_AND_dips; i++)
   {
    /* apply weighting curve around ith center frequency */
    dummy_int = rect2(ndata, f, tar_m, work_m, peaks_AND_dips[i]);

    /* find best Q and alpha for this dip */
    dummy_int = opt_sec3(ndata, f, work_m, peaks_AND_dips[i], &Q,
                     &alpha);
    K = pow(exp, alpha);

    /* get frequency response estimate for this stage */
    dummy_int = approx_s(K, peaks_AND_dips[i], Q, ndata, f, app_sec);

    if ((sse = ssel(tar_m, app_sec, ndata)) < best_sse)
    {
       best_fo = peaks_AND_dips[i];
       best_sse = sse;
       best_Q = Q;
       best_alpha = alpha;
    }
   }

   if (best_sse == DEFAULT_SSE)
   {
    printf ("\nNo solution found for stage # %d", j+1);
    exit(0);
   }
   K = pow(exp, best_alpha);

   printf ("\n\nBest K = %lf", K);
   printf ("\n\nBest Q = %lf", best_Q);
```

```
printf ("\n\nBest fo = %lf", best_fo);
/* get frequency response estimate for best stage */
dummy_int = approx_s (K, best_fo, best_Q, ndata, f, app_sec);

/* get exact frequency response  for best stage */
dummy_int = exact_s (K, best_fo, best_Q, ndata, f, exa_sec);

/*  sum (in dB)  input of this stage                    */
/* and response of this stage to get jth output          */
for (i=0; i<ndata; i++)
{
 tar_m[i] = tar_m[i] - app_sec[i];
 eq_stage_out[i] = target[i] - tar_m[i];
}

/*-------------------------------------------------------*/
/* write approximate response of jth stage to CARC file */

hdr2.Real = app_sec;
dummy_int = WritFile( &hdr2, stage_file[j]);
if ( dummy_int != 0 )
{
 PrtErr(FCT_WRITFILE, dummy_int);
}

/*-------------------------------------------*/
/*-------------------------------------------------------*/
/* write exact response of jth stage to CARC file */

hdr2.Real = exa_sec;
dummy_int = WritFile( &hdr2, exact_file[j]);
if ( dummy_int != 0 )
{
 PrtErr(FCT_WRITFILE, dummy_int);
}

/*-------------------------------------------*/

/*-------------------------------------------*/
/* write output of jth stage to CARC file   */

hdr2.Real = eq_stage_out;
dummy_int = WritFile( &hdr2, stage_out[j]);
if ( dummy_int != 0 )
{
 PrtErr(FCT_WRITFILE, dummy_int);
}


/*-----------------------------------------------*/
/* write jth filter stage specs to .log file   */

fprintf (outputf,"\n%s%d\n","Equalizer stage #",j+1);
fprintf (outputf,"%s%lf \t%s%lf \t%s%lf\n","fo = ",best_fo, "Q =
",
        best_Q, "K = ", K);
```

```
        /*------------------------------------------*/
    }

    printf ("\ngain = %lf",gain);
    exit(0);
    return(0);
}
```

Shelving Filter

```
/*************************************************************************/
/*  file adapfir3.c  :  reads in two frequency indexed CARC files,       */
/*                      one target and one measurement.                  */
/*                      Then this program uses the functions in adapt    */
/*                      to optimizive in succession N_STAGES of          */
/*                      first order shelving filters.  Then the          */
/*                      frequency response of each stage is written      */
/*                      to a CARC file for subsequent examination.       */
/*                      In addition a CARC file is written with the      */
/*                      resulting frequency response.                    */
/*                                                                       */
/*            format : ADAPFIR3 target_file measurement_file             */
/*                                                                       */
/*                                                                       */
/*  Originally coded : June 21st, 1991.                                  */
/*                                                                       */
/*  Last modified    : June 21st, 1991.                                  */
/*                                                                       */
/*************************************************************************/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <carc.h>

/* define constants    */
#define exp             2.718281828
#define DEFAULT_SSE     1000000.0
#define N_STAGES  1
#define MAX_N_STAGES    5


/* declare stack size max. */
extern unsigned _stklen=32000U;

/* function prototype declaration */
REAL64 ssel(REAL64 *targ, REAL64 *resp, INT16 ndata);

int app_fir (REAL64 K, REAL64 fo, INT16 ndata, REAL64 *f, REAL64
*resp_dB);
INT16 opt_fir3 (INT16 ndata, REAL64 *f, REAL64 *m, REAL64 gamma,
            REAL64 *best_alpha);

int exact_f (REAL64 K, REAL64 gamma, INT16 ndata, REAL64 *f, REAL64
*resp_dB);

REAL64 **matrixd(int nrows, int ncols);
REAL64 *vectord(int nrows);
int    *vectori(int nrows);
```

```
/*--------------------------------------*/

FILE *outputf;
int main (int argc, char *argv[])

{
   FHEADER hdr1; /* input data structure 1 - target file */
   FHEADER hdr2; /* input data structure 2 - measurement file */

   INT16   ndata;
   INT16   i;
   INT16   j;
   INT16   dummy_int;


   REAL64  gain;
   REAL64  *m;
   REAL64  *tar_m;
   REAL64  *target;
   REAL64  *flat;
   REAL64  *f;
   REAL64  *app_firs;
   REAL64  *eq_stage_out;
   REAL64  *exa_fir;
   REAL64  sse_min;
   REAL64  sse_max;
   REAL64  alpha;
   REAL64  K;
   REAL64  best_alpha;
   REAL64  best_sse;
   REAL64  best_fo;
   REAL64  sse;

   typedef struct {
         double K;
         double Q;
         double fo;
         } stage_id;

   stage_id stage_info[N_STAGES];
   static char *stage_file[MAX_N_STAGES] = {"fstage1.car",
                              "fstage2.car",
                              "fstage3.car",
                              "fstage4.car",
                              "fstage5.car"};

   static char *exact_file[MAX_N_STAGES] = {"fexact1.car",
                              "fexact2.car",
                              "fexact3.car",
                              "fexact4.car",
                              "fexact5.car"};

   static char *stage_out[MAX_N_STAGES] = {"firout1.car",
                             "firout2.car",
                             "firout3.car",
                             "firout4.car",
```

```
                                    "firout5.car");

   InitHead(&hdr1); /* initialize structure */
   InitHead(&hdr2); /* initialize structure */

   i = ReadFile( &hdr1, argv[1]);
   if ( i!= 0 )
   {
      PrtErr(FCT_READFILE,i);
   }

   if (hdr1.TimeFreq != D_FREQ )
   {
      printf("Input file1 is not in the frequency domain.\n");
      exit(0);
   }

   if (hdr1.Format != F_INDEXED )
   {
      printf("Input file1 format is not frequency indexed.\n");
      exit(0);
   }

   /* read second file */

   i = ReadFile( &hdr2, argv[2]);
   if ( i!= 0 )
   {
      PrtErr(FCT_READFILE,i);
   }

   if (hdr2.TimeFreq != D_FREQ )
   {
      printf("Input file2 is not in the frequency domain.\n");
      exit(0);
   }

   if (hdr2.Format != F_INDEXED )
   {
      printf("Input file2 format is not frequency indexed.\n");
      exit(0);
   }

   ndata = hdr1.NData;

   if (hdr1.NData != hdr2.NData )
   {
      printf("Input files not same length.\n");
      exit(0);
   }

   /*---------------------------------------------------*/
   /* memory allocation                                */

   tar_m           = vectord(ndata);
   eq_stage_out    = vectord(ndata);
   flat            = vectord(ndata);
```

```
app_firs = vectord(ndata);
exa_fir        = vectord(ndata);


/*--------------------------------------------------*/
/* initialize variables                           */

m      = (REAL64 *) hdr2.Real;
target = (REAL64 *) hdr1.Real;
f      = (REAL64 *) hdr2.Index;
gain = 0.0;


/*--------------------------------------------------*/
/* subtract target and initialize flat            */

for (i=0; i<ndata; i++)
{
   printf ("\nm[%d] = %lf",i, m[i]);
   tar_m[i]     = target[i] - m[i];
   flat[i] = 0.0;
}

/* open file for log data */

   if ((outputf=fopen("adapfir3.log","w")) == NULL)
   {
    printf("\n\nError opening adapfir3.log for writing.\n");
    exit(0);
   }


/*--------------------------------------------------*/
/* Find best least squares approximation one stage */
/* at a time.                                 */

for (j=0; j<N_STAGES; j++)
{
   best_fo = 0.0;
   best_alpha = 0.0;
   best_sse = DEFAULT_SSE;


   /*------------------------------------------------------------*/


   printf ("\nOptimizing stage # %d", j+1);


   /* find best shelving filter and keep best */
   for (i=0; i<ndata; i++)
   {

    dummy_int =  opt_fir3 (ndata, f, tar_m, f[i], &alpha);

    K = pow(exp, alpha);
```

```
        /* get frequency response estimate for this stage */
        dummy_int = app_fir(K, f[i], ndata, f, app_firs);

        if ((sse = ssel(tar_m, app_firs, ndata)) < best_sse)
        {
           best_fo = f[i];
           best_sse = sse;
           best_alpha = alpha;
        }

        printf ("\nPivot frequency = %lf, sse = %lf", f[i], sse);
       }

       if (best_sse == DEFAULT_SSE)
       {
        printf ("\nNo solution found for stage # %d", j+1);
        exit(0);
       }
       K = pow(exp, best_alpha);

       printf ("\n\nBest K = %lf", K);
       printf ("\n\nBest fo = %lf", best_fo);
       /* get frequency response estimate for best stage */


       dummy_int = app_fir (K, best_fo, ndata, f, app_firs);

       /* get exact frequency response  for best stage */
       dummy_int = exact_f (K, best_fo, ndata, f, exa_fir);

       /*  sum (in dB)  input of this stage                  */
       /* and response of this stage to get jth output        */
       for (i=0; i<ndata; i++)
       {
        tar_m[i] = tar_m[i] - app_firs[i];
        eq_stage_out[i] = target[i] - tar_m[i];
       }

       /*----------------------------------------------------*/
       /* write approximate response of jth stage to CARC file */

       hdr2.Real = app_firs;
       dummy_int = WritFile( &hdr2, stage_file[j]);
       if ( dummy_int != 0 )
       {
        PrtErr(FCT_WRITFILE, dummy_int);
       }

       /*----------------------------------------*/
       /*----------------------------------------------------*/
       /* write exact response of jth stage to CARC file */

       hdr2.Real = exa_fir;
       dummy_int = WritFile( &hdr2, exact_file[j]);
       if ( dummy_int != 0 )
       {
        PrtErr(FCT_WRITFILE, dummy_int);
```

```
        }

        /*-----------------------------------------*/

        /*-----------------------------------------*/
        /* write output of jth stage to CARC file   */

        hdr2.Real = eq_stage_out;
        dummy_int = WritFile( &hdr2, stage_out[j]);
        if ( dummy_int != 0 )
        {
         PrtErr(FCT_WRITFILE, dummy_int);
        }

        /*--------------------------------------------*/
        /* write jth filter stage specs to .log file   */

        fprintf (outputf,"\n%s%d\n","Equalizer stage #",j+1);
        fprintf (outputf,"%s%lf \t%s%lf \t%s%lf\n","fo = ",best_fo, "K =
", K);

        /*-----------------------------------------*/
    }

    printf ("\ngain = %lf",gain);
    exit(0);
    return(0);
}
```

Warping

```
/************************************************************************/
/*     file warp1.c   :  reads in a frequency indexed CARC file and    */
/*                       applies to the frequency index the bilinear   */
/*                       transform prewarping function.                */
/*                       The result is output to another file as       */
/*                       specified in the command line.                */
/*                                                                     */
/*            format : WARP1 infile outfile                            */
/*                                                                     */
/*                                                                     */
/*   Originally coded : April 26th, 1991.                              */
/*                                                                     */
/*   Last modified    : April 30th, 1991.                              */
/************************************************************************/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <carc.h>
/* define constants    */

#define PI              3.141592654
#define SAMP_FR     44100          /* sampling frequency    */
#define TWO_PI          6.283185308
#define INF_FREQ    99999


/* declare stack size max. */
extern unsigned _stklen=64000U;


FILE *outputf;
int main (int argc, char *argv[])

{
FHEADER hdr1; /* input/output data structure */

/* FHEADER is comprised of seven variables plus four pointers */

REAL64 arg;       /* temporary storage for argument of tan to check PI/2
*/
REAL64 *freq_ptr;
INT32 ndata;
int i;  /*  */

InitHead(&hdr1); /* initialize structure */

   i = ReadFile( &hdr1, argv[1]);
   if ( i!= 0 )
   {
      PrtErr(FCT_READFILE,i);
```

```
    }

    if (hdrl.TimeFreq != D_FREQ )
    {
       printf("Input file is not in the frequency domain.\n");
       exit(0);
    }

    if (hdrl.Format != F_INDEXED )
    {
       printf("Input file format is not frequency indexed.\n");
       exit(0);
    }

    ndata = hdrl.NData;
    freq_ptr = (REAL64 *) hdrl.Index; /* assign pointer to index pointer
in */
                                 /* in FHEADER structure for two reasons */
                                 /* 1. to save computations  */
                                 /* 2. type casting          */

/* ---------------------- */
/* frequency warping code */

    for (i=0; i++<ndata;)
    {

       arg = PI * *freq_ptr/SAMP_FR;
       if (arg==PI/2)
       {
        *freq_ptr++ = INF_FREQ;
       }
       else
       {
        *freq_ptr++ = SAMP_FR/PI * tan (arg);
       }
    }

/* ---------------------- */

    AddText (&hdrl,"WARP1.C, M. E. Bonneville, April   , 1991.");

/* writing input array with warped frequency to output CARC file */
/*
/* The response is not altered by frequency warping. */
/* However, the frequency index is altered and takes on new value */

    i = WritFile( &hdrl, argv[2]);
    if ( i!= 0 ) {
       PrtErr(FCT_WRITFILE,i);
    }

    exit(0);
    return(0);
}
```

Dewarping

```
/***************************************************************************/
/*     file dewarp1.c:  reads in a frequency indexed CARC file and        */
/*                      applies to the frequency index the bilinear       */
/*                      transform dewarping function.                     */
/*                      The result is output to another file as           */
/*                      specified in the command line.                    */
/*                                                                        */
/*            format : DEWARP1 infile outfile                             */
/*                                                                        */
/*                                                                        */
/*  Originally coded : Aug. 20th, 1991.                                   */
/*                                                                        */
/*  Last modified    : Aug. 20th, 1991.                                   */
/***************************************************************************/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <carc.h>
/* define constants    */

#define PI              3.141592654
#define SAMP_FR      44100          /* sampling frequency    */
#define TWO_PI          6.283185308


/* declare stack size max. */
extern unsigned _stklen=64000U;


FILE *outputf;
int main (int argc, char *argv[])

{
FHEADER hdr1; /* input/output data structure */

/* FHEADER is comprised of seven variables plus four pointers */

REAL64 arg;        /* temporary storage for argument */
REAL64 *freq_ptr;
INT32 ndata;
int i;  /*  */

InitHead(&hdr1); /* initialize structure */

   i = ReadFile( &hdr1, argv[1]);
   if ( i!= 0 )
   {
      PrtErr(FCT_READFILE,i);
   }

   if (hdr1.TimeFreq != D_FREQ )
   {
```

```
      printf("Input file is not in the frequency domain.\n");
      exit(0);
   }

   if (hdr1.Format != F_INDEXED )
   {
      printf("Input file format is not frequency indexed.\n");
      exit(0);
   }

   ndata = hdr1.NData;
   freq_ptr = (REAL64 *) hdr1.Index; /* assign pointer to index pointer
in */
                              /* in FHEADER structure for two reasons */
                              /* 1. to save computations   */
                              /* 2. type casting           */

/* ---------------------- */
/* frequency warping code */

   for (i=0; i++<ndata;)
   {

      arg = PI * *freq_ptr;
      *freq_ptr++ = SAMP_FR/PI * atan2(arg,SAMP_FR);
   }

/* ---------------------- */

   AddText (&hdr1,"DEWARP1.C, M. E. Bonneville, Aug.   , 1991.");

/* writing input array with warped frequency to output CARC file */
/*
/* The response is not altered by frequency warping. */
/* However, the frequency index is altered and takes on new value */

   i = WritFile( &hdr1, argv[2]);
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }

   exit(0);
   return(0);
}
```

Target Frequency Response

```
/*
/*   file target1.c  :  reads in a frequency indexed CARC file and   */
/*                      uses the same frequency indexes to interogate*/
/*                      user about target response at each of those  */
/*                      frequencies.  The target result is then      */
/*                      written to CARC file (outfile) specified in  */
/*                      command line.                                */
/*                                                                   */
/*            format : TARGET1 readfile outfile                      */
/*                                                                   */
/*                                                                   */
/*  Originally coded : April 30th, 1991.                             */
/*                                                                   */
/*  Last modified    : May    1st, 1991.                             */
/*                                                                   */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <carc.h>
/* define constants    */

#define PI                  3.141592654
#define SAMP_FR         44100          /* sampling frequency    */
#define TWO_PI              6.283185308
#define INF_FREQ        999999
#define STOP            999.0


/* declare stack size max. */
extern unsigned _stklen=64000U;


FILE *outputf;
int main (int argc, char *argv[])

{

char inbuf[10];
FHEADER hdrl; /* input/output data structure */

/* FHEADER is comprised of seven variables plus four pointers */

REAL64 arg;       /* temporary storage for argument of tan to check PI/2
*/
REAL64 *freq_ptr;
REAL64 *real_ptr;
REAL64 *imag_ptr;
REAL64 input;
INT32 ndata;
int i;  /*  */
```

```
InitHead(&hdrl); /* initialize structure */

   i = ReadFile( &hdrl, argv[1]);
   if ( i!= 0 )
   {
      PrtErr(FCT_READFILE,i);
   }

   if (hdrl.TimeFreq != D_FREQ )
   {
      printf("Input file is not in the frequency domain.\n");
      exit(0);
   }

   if (hdrl.Format != F_INDEXED )
   {
      printf("Input file format is not frequency indexed.\n");
      exit(0);
   }

   ndata = hdrl.NData;
   freq_ptr = (REAL64 *) hdrl.Index; /* assign pointer to index pointer
in */
                              /* in FHEADER structure for two reasons */
                              /* 1. to save computations  */
                              /* 2. type casting          */
   real_ptr = (REAL64 *) hdrl.Real; /* assign pointer to real pointer
*/
   imag_ptr = (REAL64 *) hdrl.Imag; /* assign pointer to imag pointer
*/

/* ---------------------- */
/* Target reading code    */
   printf ("\n\n\n           ************************************");
   printf ("\n          *** Enter 999 to exit at any time ***.\n");
   printf ( "          ************************************\n\n\n");

   for (i=0; i++<ndata;)
   {
      printf ("\n Please enter target response at %lf Hz in dB:
",*freq_ptr++);

      scanf("%lf", &input);
      printf("
%lf",input);
      *real_ptr++ = input;
      if (input == STOP)
      {
       exit(0);
      }
      *imag_ptr++ = 0.0;
   }

   printf ("\nEnd of data entry.\n");
   AddText (&hdrl,"TARGET1.C, M. E. Bonneville, May   , 1991.");
```

```
/* ------------------------------------------- */
/*                                             */
/* Write target frequency response to CARC file */
/* using same header part as measurement file   */
/* except that frequency response is replaced   */
/* with target frequency response.              */


/*   hdrl.Precision = P_REAL64;  */

   i = WritFile( &hdrl, argv[2]);
   if ( i!= 0 ) {
      PrtErr(FCT_WRITFILE,i);
   }
/*                                             */
/* ------------------------------------------- */

   exit(0);
   return(0);
}
```

Functions

```
/**********************************************************************/
/* function approx_f.c  :  approximates frequency response of         */
/*                         a first order shelving filter for given    */
/*                         pivot frequency and gain                   */
/*                         using first term of series approximation   */
/*                         in dB.                                     */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*       returns : 0                                                  */
/*               : *resp_dB = approx. filter response in dB           */
/*                                                                    */
/*                                                                    */
/*       format : approx_f (K, gamma, ndata, f, m)                    */
/*                                                                    */
/*                                                                    */
/*  Originally coded : June 21st, 1991.                               */
/*                                                                    */
/*  Last modified    : June 21st, 1991.                               */
/*                                                                    */
/*  Marc E. Bonneville                                                */
/*                                                                    */
/*                                                                    */
/***********************************************************************
/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <carc.h>


#define NEPERS_to_dB  8.68589

int approx_f (K, gamma, ndata, f, resp_dB)

REAL64 K;          /* gain at center frequency */
REAL64 gamma;            /* pivot frequency */
REAL64 Q;          /* filter quality factor */
INT16  ndata;            /* number of frequency points */
REAL64 *f;               /* pointer to frequency index array */
REAL64 *resp_dB;         /* pointer to calculated freq. resp. array */

{

/* local variables */

INT16  i;                 /* running index for matrix manipulation */
REAL64 theta;               /* frequency dependent temporary storage  */

/* ------------------------------------ */
```

```
for (i=0; i<ndata;i++)
{
   theta = Q * (f[i]/fo - fo/ f[i]);
   resp_dB[i] = 2.0 * NEPERS_to_dB *log(K) / (1 + theta*theta);
}

return(0);
}
```

```
/***********************************************************************/
/* function approx_s.c  :  approximates frequency response of         */
/*                          a second order boost/cut biquad for given */
/*                          given filter center frequency, Q and gain */
/*                          using first term of series approximation  */
/*                          in dB.                                    */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*        returns : 0                                                 */
/*                : *resp_dB = approx. filter response in dB          */
/*                                                                    */
/*                                                                    */
/*        format : approx_s (K, fo, Q, ndata, f, m)                   */
/*                                                                    */
/*                                                                    */
/*  Originally coded : May 7th, 1991.                                 */
/*                                                                    */
/*  Last modified    : May 7th, 1991.                                 */
/*                                                                    */
/*  Marc E. Bonneville                                                */
/*                                                                    */
/*                                                                    */
/***********************************************************************
/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <carc.h>


#define NEPERS_to_dB  8.68589

int approx_s (K, fo, Q, ndata, f, resp_dB)

REAL64 K;          /* gain at center frequency */
REAL64 fo;              /* center frequency */
REAL64 Q;          /* filter quality factor */
INT16  ndata;           /* number of frequency points */
REAL64 *f;              /* pointer to frequency index array */
REAL64 *resp_dB;        /* pointer to calculated freq. resp. array */

{

/* local variables */

INT16  i;                /* running index for matrix manipulation */
REAL64 theta;              /* frequency dependent temporary storage  */

/* ------------------------------------ */


for (i=0; i<ndata;i++)
{
   theta = Q * (f[i]/fo - fo/ f[i]);
```

```
   resp_dB[i] = 2.0 * NEPERS_to_dB *log(K) / (1 + theta*theta);
}

return(0);
}
```

```
/*                                                                        */
/* function centr_f1.c  :  estimates the center frequencies.  This is*/
/*                         done by taking the derivative of the mag. */
/*                         response in dB and noting a sign change   */
/*                         The derivative is approximated using a    */
/*                         backward difference equation.             */
/*                                                                   */
/*                                                                   */
/*            format : centr_fr ()                                   */
/*                                                                   */
/*                                                                   */
/*  Originally coded : June 4th, 1991.                               */
/*                                                                   */
/*  Last modified    : June 5th, 1991.                               */
/*                                                                   */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <carc.h>

#define  STOP    0
#define  CONT    1

/* function prototype declarations      */
REAL64 sign(REAL64 der);
REAL64 *vectord(INT16 ndata);
REAL64 norm_min (INT16 ndata, REAL64 *resp_dB);

/* ------------------------------------ */

INT16 centr_f1(ndata, f, m, cent_fo, der)


INT16       ndata;            /* number of frequency points */
REAL64      *f;         /* frequency vector pointer */
REAL64      *m;               /* frequency response vector pointer*/
REAL64      *cent_fo;   /* center frequencies vector pointer*/
REAL64   *der;          /* derivative vector pointer*/

{

   /* local variables */

   INT16  i;            /* running index for frequencies */
   INT16  j;            /* running index for frequencies */
   REAL64 norm;         /* temporary ausigment */
   INT16  count_fo;     /* number of center frequencies */
   REAL64 last_diff_sign_f;   /* last frequency at which sign changed */
   REAL64  *new_m;      /* modified m for use when taking derivative */
   REAL64 last_diff_sign;     /* last before it changed but not zero */
   REAL64 ratio;        /* temporary storage */
   REAL64 log_ratio;          /* temporary storage */
   INT16  flag;


   /* ---------------------------------------------------- */
```

```
   /* memory allocation for local pointers              */

   new_m = vectord(ndata);


   for (i=0;i<ndata;i++)
   {
      new_m[i] = m[i];
   }


   norm = norm_min (ndata,new_m);
   for (i=0; i<ndata;i++)
   {
      new_m[i] = new_m[i] + 1.0;

   }

   printf ("\nTaking derivative\n");
   der[0]=0.0;
   for (i=1;i<ndata;i++)
   {
      der[i] = (new_m[i] - new_m[i-1]);  /* backward difference eq'n */
/*    der[i] = (new_m[i] - new_m[i-1] - der[i-1]);  bilinear  ? */
   }


   printf ("\nFind first non zero derivative\n");

   /* ------------------------------ */
   /* find first non zero derivative */
   flag=CONT;
   i=0;
   while  (flag==CONT)
   {
      if (i<ndata)
      {
       if (sign(der[i])!= 0)
       {
          last_diff_sign   = sign(der[i]);
          last_diff_sign_f = f[i];
          flag=STOP;
       }
      }
      else
      {
       /* all derivatives are zero */
       return(0);
      }
      i++;
   }
   /* ------------------------------ */

   /* ------------------------------ */
   /* center frequencies           */
   count_fo = 0;
   printf ("\n Looking for center frequencies.\n");
```

```
   for (i=2;i<ndata;i++)
   {
      if (der[i]!=0.0)
      {
       if (sign(der[i]) != last_diff_sign)
       {
          /* take logarithmic average frequency */
/*        printf("\nf[%d] = %lf\n", i, f[i]); */
/*        printf("\nlast_diff_sign_f = %lf\n", last_diff_sign_f); */
          ratio = f[i] / last_diff_sign_f;
          printf ("\nm = %lx\n",(long) m);
          printf ("\nnew_m = %lx\n",(long) new_m);
          printf ("\nf = %lx\n",(long) f);
          printf ("\ncent_fo = %lx\n",(long) cent_fo);
          printf ("\nder = %lx\n",(long) der);

          printf ("\n check 1, ratio = %lf\n",ratio);
          log_ratio = log10(ratio);
          printf ("\n check 2");
          cent_fo[count_fo] = last_diff_sign_f *
pow(10.0,log_ratio/2.0);
          printf ("\n check 3");
          count_fo = count_fo + 1;
/*        printf ("\n check 4, count_fo = %d\n",count_fo); */
          last_diff_sign = sign(der[i]);
/*        printf ("\n check 5"); */
          last_diff_sign_f = f[i];
       }
       else
       {
/*        printf("\nfirst statement in else\n"); */

          last_diff_sign = sign(der[i]);
          last_diff_sign_f = f[i];
       }
      }
   }

/*  for (i=0; i<count_fo; i++)                          */
/*  {                                                    */
/*     printf ("\ncent_fo[%d] = %lf",i,cent_fo[i]);     */
/*  }                                                    */

     printf ("\nfunction centr_f1 : count_fo = %d\n",count_fo);


   free(new_m);

   return(count_fo);
}

REAL64 sign(REAL64 a)
{
   if (a>0.0)
   {
```

```
      return(1.0);
   }
   if (a<0.0)
   {
      return(-1.0);
   }
   return(0.0);

}
```

```
/**********************************************************************/
/* function exact_f.c   :  calculates exact frequency response of    */
/*                          a shelving filter                         */
/*                          given filter pivot frequency and gain.    */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*       returns : 0                                                  */
/*               : *resp_dB =  filter response in dB                  */
/*                                                                    */
/*                                                                    */
/*       format : exact_f (K, fo, Q, ndata, f, m)                     */
/*                                                                    */
/*                                                                    */
/*  Originally coded : June 21st, 1991.                               */
/*                                                                    */
/*  Last modified    : June 21st, 1991.                               */
/*                                                                    */
/*  Marc E. Bonneville                                                */
/*                                                                    */
/*                                                                    */
/**********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <carc.h>



int exact_f (K, gamma, ndata, f, resp_dB)

REAL64 K;          /* gain at center frequency */
REAL64 gamma;      /* pivot frequency */
INT16  ndata;           /* number of frequency points */
REAL64 *f;              /* pointer to frequency index array */
REAL64 *resp_dB;        /* pointer to calculated freq. resp. array */

{

/* local variables */

INT16  i;                /* running index for matrix manipulation */
REAL64 theta;              /* frequency dependent temporary storage  */
REAL64 arg;                /* temporary storage  */

/* ------------------------------------ */


for (i=0; i<ndata;i++)
{
   theta = f[i]/gamma;
   arg   = (theta*theta + K*K) /
         (theta*theta + 1.0 / (K*K) );
   resp_dB[i] = 10.0 * log10(arg);
}
```

```
return(0);
}
```

```
/**********************************************************************/
/* function exact_s.c   :  calculates exact frequency response of     */
/*                         a second order boost/cut biquad for given   */
/*                         given filter center frequency, Q and gain. */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*         returns : 0                                                */
/*                 : *resp_dB =  filter response in dB                */
/*                                                                    */
/*                                                                    */
/*         format : exact_s (K, fo, Q, ndata, f, m)                   */
/*                                                                    */
/*                                                                    */
/*  Originally coded : June 19th, 1991.                               */
/*                                                                    */
/*  Last modified    : June 19th, 1991.                               */
/*                                                                    */
/*  Marc E. Bonneville                                                */
/*                                                                    */
/*                                                                    */
/**********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <carc.h>



int exact_s (K, fo, Q, ndata, f, resp_dB)

REAL64 K;          /* gain at center frequency */
REAL64 fo;               /* center frequency */
REAL64 Q;          /* filter quality factor */
INT16  ndata;            /* number of frequency points */
REAL64 *f;               /* pointer to frequency index array */
REAL64 *resp_dB;         /* pointer to calculated freq. resp. array */

{

/* local variables */

INT16  i;                 /* running index for matrix manipulation */
REAL64 theta;               /* frequency dependent temporary storage  */
REAL64 arg;                 /* temporary storage  */

/* ------------------------------------ */


for (i=0; i<ndata;i++)
{
   theta = Q * (f[i]/fo - fo/ f[i]);
   arg   = (theta*theta + K*K) /
         (theta*theta + 1.0 / (K*K) );
   resp_dB[i] = 10.0 * log10(arg);
```

```
}

return(0);
}
```

```
/****************************************************/
/*                                                  */
/* matrixd.c  : allocate memory for an n x m matrix, */
/*              double precision and initialize     */
/*              elements to zero.                   */
/*                                                  */
/*                                                  */
/* Ref. : Numerical Recipes in C, 1988, p.706       */
/*                                                  */
/* in program that calls this function : declare    */
/* matrix as a pointer to pointer                   */
/*                                                  */
/* ex. :  double **H;                               */
/*              .                                   */
/*              .                                   */
/*              .                                   */
/*                                                  */
/*        H = matrixd(#of rows,#of cols);           */
/*                                                  */
/****************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <carc.h>



double **matrixd(n,m)
int n,m;
{
   int i, j;
   double **p;


printf ("\nFunction MATRIXd: n= %d\n",n);
printf ("\nFunction MATRIXd: m= %d\n",m);

   /* allocate pointers to rows */

   if ((p=(double **)malloc((unsigned)n*sizeof(double *))) == NULL)
   {
      printf("Memory allocation failure 1\n");
      printf("p= %ld\n",p);
      return(1);
   }

   /* allocate rows and set pointers to them */
   for (i=0; i<n;i++)
   {
      if ((p[i] = (double *) malloc((unsigned)m*sizeof(double))) ==
NULL)
      {
       printf("Memory allocation failure 2\n");
       return(1);
      }
   }
```

```
   for (i=0;i<n;i++)
   {
      for (j=0;j<m;j++)
      {
       p[i][j] = 0.0;
      }
   }
   return(p);
}
```

```
/*                                                                    */
/* function opt_fir3.c  : finds best least squares alpha for          */
/*                        first order boost/cut shelving filter for a*/
/*                        given frequency response and given filter  */
/*                        pivot frequency (fo).                       */
/*                                                                    */
/*       returns : 1 = no solution found                              */
/*               : 0 = solution found                                 */
/*               : REAL64 *best_alpha                                 */
/*               : REAL64 *best_Q                                     */
/*                                                                    */
/*                                                                    */
/*                                                                    */
/*       format : opt_fir3 (ndata, *f, *m, gamma,  *best_alpha)      */
/*                                                                    */
/*                                                                    */
/*  Originally coded : June 21st, 1991.                               */
/*                                                                    */
/*  Last modified    : June 21st, 1991.                               */
/*                                                                    */
/*  Marc E. Bonneville                                                */
/*                                                                    */
/*                                                                    */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <carc.h>


/* define constants */
#define NEPERS_to_dB              8.68589

/* ------------------------------------ */
/* function prototype declarations      */


/* ------------------------------------ */



INT16 opt_fir3 (ndata, f, m, gamma, best_alpha)

INT16 ndata;              /* number of frequency points */
REAL64 *f;                /* pointer to frequency index array */
REAL64 *m;                /* pointer to given  frequency response */
REAL64 gamma;             /* pivot frequency */
REAL64 *best_alpha;       /* best least squares alpha = ln(K) */

{

   /* local variables */

   INT16  i;                   /* running index for matrix manipulation
*/
   REAL64 alpha;          /* optimum alpha value for given Q */
```

```
   REAL64 sum_num;            /* temporary storage for optimum alpha
calculation */
   REAL64 sum_den;            /* temporary storage for optimum alpha
calculation */
   REAL64 x;                  /* temporary storage */
   REAL64 temp;               /* temporary storage */


   /* ---------------------------------------------------- */
   /* memory allocation for local pointers              */


   /* ---------------------------------------------------- */

   /*--------------------------------------------------------*/
   /*                                                        */



   sum_num = 0.0;
   sum_den = 0.0;

   /*--------------------------------------------*/
   /* optimum alpha calculation for given gamma  */

   for (i=0; i<ndata;i++)
   {
      x = f[i] / gamma;
      temp = 1 +  x*x;
      sum_num += m[i] / temp;
      sum_den += 1.0 / (temp*temp);
   }
   *best_alpha = sum_num / sum_den / (2.0*NEPERS_to_dB);
   return(0);
}
```

```
/*                                                                      */
/* function opt_sec3.c  :  finds best least squares alpha and Q for    */
/*                         a second order boost/cut biquad for a given*/
/*                         frequency response and given filter center */
/*                         frequency (fo).                             */
/*                                                                      */
/*                      :  this function uses several values of Q's,   */
/*                         finds the best gain term (alpha) for each   */
/*                         and calculates the sse for each.  Then it   */
/*                         returns the Q and alpha pair giving the     */
/*                         best least squares error.                   */
/*                                                                      */
/*        returns : 1 = no solution found                               */
/*                : 0 = solution found                                  */
/*                : REAL64 *best_alpha                                  */
/*                : REAL64 *best_Q                                      */
/*                                                                      */
/*                                                                      */
/*                                                                      */
/*        format : opt_sec3 (ndata, *f, *m, fo, *best_Q, *best_alpha) */
/*                                                                      */
/*                                                                      */
/*  Originally coded : June 3rd, 1991.                                  */
/*                                                                      */
/*  Last modified    : June 4th, 1991.                                  */
/*                                                                      */
/*  Marc E. Bonneville                                                  */
/*                                                                      */
/*                                                                      */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <carc.h>



/* define constants */
#define NEPERS_to_dB            8.68589
#define ORDER                   2
#define     exp                       2.718281828
#define Q_START                 0.0
#define Q_STOP                 10.0
#define BEST_SSE_DUMMY       1000000.0
#define BEST_Q_DUMMY             -1.0
#define BEST_ALPHA_DUMMY          0.0
#define Q_STEP1                   1.0    /* Q step size for first pass
*/
#define Q_STEP2                   0.2  /* Q step size for second pass
*/

/* ------------------------------------ */
/* function prototype declarations      */
REAL64 *vectord(int nrows);

REAL64 ssel(REAL64 *m, REAL64 *filter_resp, INT16 ndata);
```

```
INT16 approx_s (REAL64 K, REAL64 fo, REAL64 Q, INT16 ndata,
            REAL64 *f, REAL64 *filter_resp);

/* ------------------------------------ */



INT16 opt_sec3 (ndata, f, m, fo, best_Q, best_alpha)

INT16 ndata;            /* number of frequency points */
REAL64 *f;              /* pointer to frequency index array */
REAL64 *m;              /* pointer to given  frequency response */
REAL64 fo;              /* center frequency */
REAL64 *best_Q;         /* best least squares Q */
REAL64 *best_alpha;     /* best least squares alpha = ln(K) */

{

   /* local variables */

   INT16  i;                  /* running index for matrix manipulation
*/
   INT16  j;                  /* running index for matrix manipulation
*/
   REAL64 Q;                  /* current filter Q value */
   REAL64 alpha;        /* optimum alpha value for given Q */
   REAL64 K;                  /* e to the power alpha */
   REAL64 *filter_resp;       /* filter response in dB */
   REAL64 sum_sq_error;       /* sum of the square errors */
   REAL64 sum_num;            /* temporary storage for optimum alpha
calculation */
   REAL64 sum_den;            /* temporary storage for optimum alpha
calculation */
   REAL64 best_sse;           /* lowest value of sum of square errors */
   REAL64 x;                  /* temporary storage */
   REAL64 temp;               /* temporary storage */


   /* --------------------------------------------------- */
   /* memory allocation for local pointers             */


/*   printf ("\nBefore Function VECTORD - filter_resp\n"); */
   filter_resp = vectord(ndata);
   printf ("\nAt start of opt_sec3 filter_resp = %lx\n",(long)
filter_resp);

/*   printf ("\nAfter function VECTORD - running optimization\n"); */

   /* --------------------------------------------------- */

   /*-------------------------------------------------------*/
   /*   first pass : do gross scan of Q's                  */
   /*                                                       */

   Q=Q_START;
   best_sse = BEST_SSE_DUMMY;
```

```
   *best_Q     = BEST_Q_DUMMY;
   *best_alpha = BEST_ALPHA_DUMMY;


   while (Q<=Q_STOP)
   {
      sum_num = 0.0;
      sum_den = 0.0;

      /*----------------------------------------*/
      /* optimum alpha calculation for given Q  */

      for (i=0; i<ndata;i++)
      {
       x = f[i] / fo  -  fo / f[i];
       temp = 1 + Q*Q * x*x;
       sum_num += m[i] / temp;
       sum_den += 1.0 / (temp*temp);
      }
      alpha = sum_num / sum_den / (2.0*NEPERS_to_dB);
      /*----------------------------------------*/

      K = pow(exp,alpha);
/*      printf (" \nK = %lf  Q = %lf\n",K,Q); */
      j = approx_s(K, fo, Q, ndata, f, filter_resp);

      sum_sq_error = ssel(m, filter_resp, ndata);
/*      printf (" sse = %lf\n",sum_sq_error); */


      if (sum_sq_error < best_sse)
      {
       best_sse = sum_sq_error;
       *best_Q = Q;
       *best_alpha = alpha;
      }
      Q += Q_STEP1;
   }
   /*-----------------------------------------------------------*/

/*   printf (" \nFirst pass best alpha = %lf  best Q =
%lf\n",*best_alpha,*best_Q); */

   if ((*best_Q - Q_STEP1) < Q_START)
   {
      Q = Q_START;
   }
   else
   {
      Q = *best_Q - Q_STEP1;
   }

   /*-----------------------------------------------------------*/
   /*   second pass : zoom in on best Q found in first pass     */
   /*                                                           */

   while (Q<=Q_STOP)
```

```
   {
      sum_num = 0.0;
      sum_den = 0.0;

      /*---------------------------------------*/
      /* optimum alpha calculation for given Q  */

      for (i=0; i<ndata;i++)
      {
       x = f[i] / fo  -  fo / f[i];
       temp = 1 + Q*Q * x*x;
       sum_num += m[i] / temp;
       sum_den += 1.0 / (temp*temp);
      }
      alpha = sum_num / sum_den / (2.0*NEPERS_to_dB);
      /*---------------------------------------*/

      K = pow(exp,alpha);
      j = approx_s(K, fo, Q, ndata, f, filter_resp);

      sum_sq_error = ssel(m, filter_resp, ndata);


      if (sum_sq_error < best_sse)
      {
       best_sse = sum_sq_error;
       *best_Q = Q;
       *best_alpha = alpha;
      }
      Q += Q_STEP2;
   }
/*   printf (" \nSecond pass best alpha = %lf  best Q =
%lf\n",*best_alpha,*best_Q); */

   /*-----------------------------------------------------------*/
   printf ("\nAt end of opt_sec3 filter_resp = %lx\n",(long)
filter_resp);

   free (filter_resp);
   if (Q == BEST_Q_DUMMY)
   {
      printf ("Unrealizable filter : Q less than zero.\n");
      return(1);
   }
   else
   {
      return(0);
   }
}
```

```
/*                                                                     */
/* function rect2.c  :     Weights data using a rectangular window.   */
/*                         The rectangular window is centered on      */
/*                         given center frequency fo and extend on    */
/*                         both sides to lo_fo and up_fo.             */
/*                         lo_fo and up_fo correspond to the two      */
/*                         adjacent zero crossings in m.              */
/*                                                                     */
/*                 :     This function is used with adapsec2 where    */
/*                     the data is not normalized or is normalized    */
/*                     to the average response.                       */
/*                                                                     */
/*    format : rect2(ndata, f, m, work_m, count_fo, cent_fo, fo)      */
/*                                                                     */
/*                                                                     */
/*  Originally coded : June 14th, 1991.                               */
/*                                                                     */
/*  Last modified    : June 14th, 1991.                               */
/*                                                                     */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <carc.h>

#define     CONT  1
#define     STOP  0

/* function prototype declaration */

REAL64 sign(REAL64 a);


INT16 rect2(ndata, f, m, work_m, fo)


INT16       ndata;           /* number of frequency points */
REAL64      *f;         /* frequency vector pointer */
REAL64      *m;              /* frequency response vector pointer*/
REAL64      *work_m;    /* windowed frequency response vector pointer*/
REAL64      fo;              /* center frequency around which window is
centered */

 {

   /* local variables */

   INT16  i;            /* running index for frequencies */
   INT16  j;            /* running index for frequencies */
   INT16   flag1;       /* outer loop status flag */
   INT16   flag2;       /* inner loop status flag */
   REAL64 lo_fo;  /* lower frequency at which m has a zero crossing */
   REAL64 up_fo;  /* upper frequency at which m has a zero crossing*/
   REAL64 sign_m_at_fo; /* sign of m at fo */


 /*   printf ("\nndata = %d", ndata);    */
```

```
/*   printf ("\nfo = %lf", fo);            */


   /* find adjacent zero crossings */
   flag1 = CONT;
   i=0;
   while (flag1 == CONT)
   {
      /* find frequency same as fo or first higher one */
      if (f[i] >= fo)
      {
       sign_m_at_fo = sign(m[i]);
/*     printf ("\nsign_m_at_fo  = %lf", sign_m_at_fo );  */

       j=i;
       flag2 = CONT;
       while (flag2 == CONT)
       {
          if (j ==(ndata-1))
          {
             up_fo = f[ndata-1];
             flag2 = STOP;
          }
          else
          {
             /* find first zero crossing above fo */
             if ((sign(m[j])) != sign_m_at_fo)
             {
              up_fo = f[j];
              flag2 = STOP;
             }
          }
          j++;
       }

       j=i;
       flag2 = CONT;
       while (flag2 == CONT)
       {
          if (j == 0)
          {
             lo_fo = f[0];
             flag2 = STOP;
          }
          else
          {
             /* find first zero crossing below fo */
             if ((sign(m[j])) != sign_m_at_fo)
             {
              lo_fo = f[j];
              flag2 = STOP;
             }
          }
          j--;
       }
       flag1 = STOP;
```

```
      }
      else
      {
       if (i == (ndata-1))
       {
          /* center frequency is higher than highest measured frequency
*/
          printf ("\n\n Function rect2 : Center frequency is higher
than\n");
          printf ("highest measured frequency.\n");
          return(1);
       }
      }
      i++;
   }

   /* rectangular weighting window */
   for (i=0; i<ndata; i++)
   {
      if ((f[i] >= lo_fo) && (f[i] <= up_fo))
      {
       work_m[i] = m[i];
      }
      else
      {
       work_m[i] = 0.0;
      }

   }

   printf ("\nlo_fo = %lf\n",lo_fo);
   printf ("\nup_fo = %lf\n",up_fo);

   return (0);
}
```

```
/*                                                                      */
/* function ssel.c  :  calculates sum of squared error between two     */
/*                     input curves.                                    */
/*                                                                      */
/*                                                                      */
/*          format : ssel (targ, resp, ndata)                           */
/*                                                                      */
/*                                                                      */
/*  Originally coded : May 2nd, 1991.                                   */
/*                                                                      */
/*  Last modified    : May 2nd, 1991.                                   */
/*                                                                      */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <carc.h>


REAL64 ssel(targ, resp, ndata)

REAL64 *targ; /* pointer to target frequency response */
REAL64 *resp; /* pointer to given  frequency response */
INT16 ndata;

{

/* local variables */

INT16  i;       /* running index for frequencies */
REAL64 diff;      /* difference between the two curves */
REAL64 sum_se;  /* counter for storing the sum of the squared errors */


sum_se =0.0;

for (i=0; i<ndata;i++)
{
   diff = targ[i] - resp[i];
   sum_se += diff * diff;   /* squared error */
}

   return(sum_se);
}
```

```
/*****************************************************/
/*                                                   */
/* vectord.c  : allocate memory for an n x 1 vector, */
/*              double precision and initialize      */
/*              elements to zero.                    */
/*                                                   */
/*                                                   */
/* Ref. : Numerical Recipes in C, 1988, p.706        */
/*                                                   */
/* in program that calls this function : declare     */
/* vector as a pointer                               */
/*                                                   */
/* ex. :  double *v;                                 */
/*              .                                    */
/*              .                                    */
/*              .                                    */
/*                                                   */
/*        v = vectord(#of elements);                 */
/*                                                   */
/*****************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <carc.h>


double *vectord(n)
int n;
{
   int i;
   double *p;



   if ((p=(double *)malloc((unsigned)n*sizeof(double))) == NULL)
   {
      printf("Memory allocation failure 1\n");
      printf("p= %ld\n",p);
      return(1);
   }

   for (i=0;i<n;i++)
       p[i] = 0.0;
   return(p);
}
```