Constrained Least-Squares Linear-Phase FIR Filter Synthesis (CLSLP)

September 28, 2024

William Earl Jones

http://www.wejc.com ‖ **wejc@wejc.com**

Last Updated 09/28/24 at 13:46

1. Introduction

This document describes use of the Constrained Least-Squares Linear-Phase FIR Filter Synthesis Algorithm CLSLP as described in reference [1]. The acronym CLSLP is not used in the technical paper and is only used as a local convenience and as a name for the software package. The acronym *LPFIR* is used here as an abbreviation for *Linear-Phase Finite Impulse Response* filter.

2. Linear-Phase FIR (LPFIR) Filters

A Linear-Phase FIR filter (LPFIR) is a special form of FIR filter that preserves the timing relationship between signals of different frequencies. Non-Linear phase FIR filters have different phase responses at different frequencies and these errors can accumulate as frequency-dependent time delays in multilayered signal-processing schemes. The linear-phase constraint guarantees that the time delay of different frequency components is identical. It's also easy to show that cascaded linear-phase FIR filters are themselves linear phase. This means complicated multi-layer filtering schemes can be employed without this cumulative phase (timing) degradation.

The CLSLP filter synthesis algorithm is computationally efficient and fast even on a standard desktop PC. For example, the synthesis of the filter in Section 3 below, a 501-coefficient real-valued filter, takes roughly *1.6 to 3* ms in total, or about *3* µs per FIR filter coefficient.

3. *Using the CLSLP Algorithm*

This Section provides illustrations of the practical use and characteristics of the Constrained Least-Squares Linear-Phase FIR Filter (CLSLP) synthesis algorithm which was presumably first described in reference [1]. CLSLP filters are minimum weighted square-error and possess linear-phase due to their conjugate-symmetric coefficient form. CLSLP filters are generated without a Fourier Transform and the fidelity of the synthesized filter coefficients is quite good (e.g. *-150* dB stop-band in Figure 1 below). The CLSLP algorithm is also computationally efficient and so is ideal for adaptive real-time filtering applications. For example, the 501-coefficient linear-phase FIR filter below was synthesized with double-precision floating-point coefficients in

1.69 ms on a standard desktop PC. The center filter segment is an *exponential-line* segment type, from .1 an .4 Normalized Hz, with amplitudes of *-80* dB and *0* dB at the left and right filter segment edges respectively. The other frequency segments are set to zero. Note, exponential-filter segments appear as straight lines on dB plots.
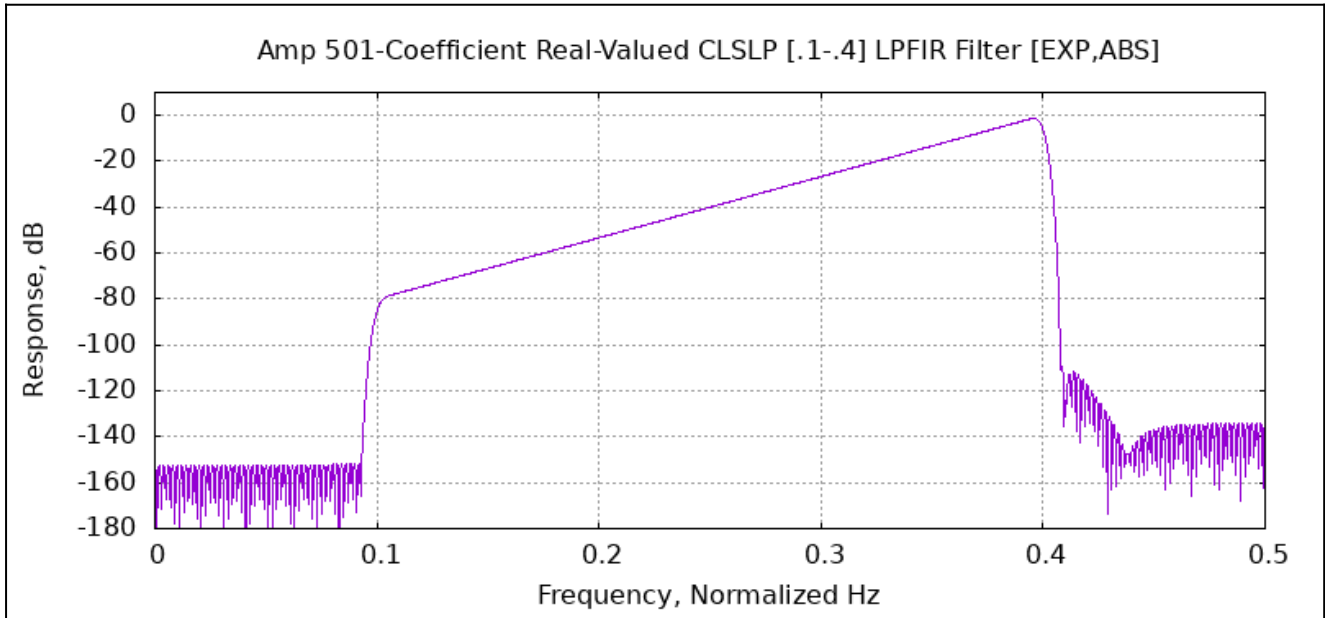

Figure 3.1: 501-Coefficient Real-Valued CLSLP [.1-.4] LPFIR Filter [EXP,ABS].

3.1. Filter Frequency-Interval Definitions

CLSLP filter coefficients are given in Equations (32) and (33) in reference [1]. Breaking frequency up into sub-bands is intuitive as (32) and (33) are integrals over frequency. This easily accommodates unspecified frequency bands as well. By judiciously choosing the amplitude response functions used for these intervals, filter synthesis can be achieved quickly resulting in high-precision LPFIR filter coefficients.

3.1.1. Frequency-Interval Types

There are currently four types of frequency intervals defined. These intervals are either *linear* or *exponential*. An *exponential* interval appears as a straight line on a dB plot while a *linear* interval appears as a straight line on a linear plot. Additionally these segments employ either *absolute* or *relative* weighting.

Absolute Weighting Type	Normal square-error
Relative Weighting **Type**	Normal square-error divided by the specified signal amplitude. This mode artificially boosts the fit quality of lower amplitude frequency intervals. Signal amplitudes of zero cannot be used as logarithms are employed. Use a small value instead (e.g. *1e-8*).

This is the C++ data structure specification of frequency interval types

```
// Segment types
//  [F0,F1] in frequency and [A0,A1] in amplitude for Linear Segments
//  [F0,F1] in frequency and [loge(A0),loge(A1)] in amplitude for Exponential Segments
//  Note that frequencies are specified in the interval [0,1) Normalized Hz.
typedef enum  eSegType
{
  eExp_RelErr, // exp(B*f+A) segment, relative error
  eExp_AbsErr, // exp(B*f+A) segment, absolute error
  eLin_RelErr, // B*f+A segment, relative error
  eLin_AbsErr  // B*f+A segment, absolute error
} LS_FIR_Filter;
```

Due to the open form of this algorithm, additional frequency-interval types can be added as needed. These four frequency-interval types are computationally efficient and have, so far anyway, sufficed in practical filter-design problems.

3.1.2. Segment Parameters

Five numeric values are needed for any of the four segment types mentioned in Section 3.1.1 above. Namely *X[0], X[1], Y[0], Y[1],* and the weighting factor which is discussed in the next section. Here *X[0]* and *X[1]* are interval start and stop frequencies given in *Normalized Hz*. The filter amplitudes at the frequency-interval edges are specified as Y[0] and Y[1] (not in dB). The frequency interval type is Linear/Absolute error *LS_FIR_Filter::eLin_AbsErr*.

3.1.3. Segment Weights

Segment weights are provided so the filter designer can change the fit quality filter sections. A value of *1* is considered normal weighting, a value of *0* means the fit error for that section is completely ignored, and a segment weight of *2* doubles the fit error. Increasing the segment weight should improve the fit in that filter segment. For example, the middle filter segment of the filter illustrated in the previous section would be specified as *X[0]=.1, X[1]=.4, Y[0]=1e-4, Y[1]=1, Type=LS_FIR_Filter::eLin_AbsErr, Weight=1* .

3.2. Using CLSLP Software

This section illustrates installing, building, then running the CLSLP filter synthesis application.

3.3. Installation

It is fairly easy to install CLSLP and synthesize filters on any platform or context. There is no installation procedure per se. CLSLP is released here as C++20 source code and CMAKE build files. Additionally Linux command files *CLEAN* and *BUILD_AND_RUN* provide illustration of the proper command syntax. This software should build on most modern operating systems and CPU architectures. Though Linux has been used in this software's design, MS Windows, Mac OS, and many other operating systems can be used as both the build and run environment. Build and run environments can be set separately via compiler switches in modern C++ compilers so heterogeneous computer operating systems and architectures can be integrated easily.

Executing *BUILD_AND_RUN* should do everything in one step. The *CLEAN* command isn't really necessary unless CMAKE gets confused. *CLEAN* empties the build directory completely.

```
./CLEAN              # Not really necessary --- clears subdir build/
./BUILD_AND_RUN      # Build then Run the software
```

3.4. CLSLP Software Distribution

CLSLP is distributed in C++ source code form. CMAKE build files are included as well as Linux script files to illustrate proper use.

3.4.1. CLSLP Software Distribution

The standard CLSLP release consists of, at least, the files listed at the end of this section. The Linux GSL special function library is necessary for CLSLP LPFIR filter synthesis. FFTW and GNUPLOT are used for plot generation though neither is needed for coefficient synthesis itself.

A CLSLP release will contain *at least* the files specified below

- **Start.cpp**
- **CLSLP.h**
- **CLSLP.cpp**
- **AlignedMemAlloc.h**
- **CLEAN**
- **BUILD_AND_RUN**
- **GENERATE_PLOTS**

Header file *AlignedMemAlloc.h* helps provide functionality for SSE and AVX SIMD vector allocation (faster execution). This module is also the C++ *main* program entry-point for the software in these tests. As a local convention we use *start* as part of the *main's* filename to identify it. Filename like *Start_x.cpp* are used for C++ *main* files in this document. For example of use see Section 4.1.2 below.

3.4.2. Building Application with CMAKE

Execute the following commands in the source code where you want to store your build files. The sub-directory *build* is commonly used.

```
./CLEAN            # Cleans the build directory  (optional)
./BUILD_AND_RUN    # Builds (CMAKE) source code and execute program
./GENERATE_PLOTS   # Generate GNUPLOT plots from data files just created (optional)
```

3.4.3. Spectral Plots

The CMAKE current directory should now contain two new PNG graphics files: the Amplitude and Phase Response graphs.

3.5. CLSLP Filter Synthesis Applications

The only module we need to write is the *start-file,* or file *Start_x.cpp*. This is the C++ main that calls the CLSLP filter synthesis software. *GenFilter* is then called to synthesize the filter coefficients. The code fragment below synthesizes an LPFIR Filter with 501 real-valued double-precision coefficients into output vector *OutCoefs*. This vector is allocated on AVX (SIMD) memory boundaries. On Linux systems, *-O3 x5pT=rAu(optimized compilation),* is sufficient for basic SIMD support on CPUs that support it. Intel and AMD PC CPUs normally do support it.

```cpp
// The C++ code segment that generates this 501-coefficient CLSLP filter
#include "CLSLP.h"

// Generate the CLSLP filter Coefficients into OutCoefs
const unsigned int     NumFiltCoefs = 501;
CLSLP                  Filt;
vector<complex<double>>  OutCoefs(NumFiltCoefs); // Real-valued Coefs stored in Complex

// 'AddSymmetric' is for Real-Valued Filter Coefficients    ([0,.5] mod 1.) Normalized Hz
// 'Add'          is for Complex-Valued Filter Coefficients ([0,1.) mod 1.) Normalized Hz
Filt.AddSymmetric( LS_FIR_Filter::eLin_AbsErr, .0, .1,  0.00,  0.00,  1 );
Filt.AddSymmetric( LS_FIR_Filter::eExp_RelErr, .1, .4,  1e-4,  1.00,  1 );
Filt.AddSymmetric( LS_FIR_Filter::eLin_AbsErr, .4, .5,  0.00,  0.00,  1 );

// Calculate the FIR Filter Coefficients into OutCoefs
Filt.GenFilter( NumFiltCoefs, OutCoefs );
```

3.6. Building Filter Synthesis Software

Next we compile and link the software in the previous section into an executable. We are using CMAKE and the GNU C++ compiler build tools. The following BASH script performs the CMAKE build.

```bash
#! /bin/bash

# Generate the CMAKE files
make --fresh   -S .  -B build  -DCMAKE_BUILD_TYPE=RELEASE  -DCMAKE_VERBOSE_MAKEFILE=OFF

# Compile the C++ sources
cmake  --build  build
```

A listing of the CMAKE control file *CMakeLists.txt* is presented below as a convenience.

```cmake
cmake_minimum_required(VERSION 3.10)

# specify the C++ standard
set(CMAKE_BUILD_TYPE Release)
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_FLAGS_RELEASE "-O3 -Wall -Wextra")
set(CMAKE_CXX_FLAGS_DEBUG "-g -Wall -Wextra")
# set(CMAKE_VERBOSE_MAKEFILE ON)

# Set some basic project attributes
project (CLSLP
         VERSION  1.00
         DESCRIPTION "CLSLP Filter Synthesis")

file(GLOB SRC_FILES ${PROJECT_SOURCE_DIR}/*.cpp)

# This project will output an executable file
add_executable(${PROJECT_NAME} ${SRC_FILES})

# Include the configuration header in the build
target_include_directories(${PROJECT_NAME} PUBLIC "${PROJECT_BINARY_DIR}")

# Math includes
target_link_libraries(${PROJECT_NAME} PUBLIC "gsl")
target_link_libraries(${PROJECT_NAME} PUBLIC "gslcblas")
```

The following sequence of commands will perform the C++ build, then execute the synthesized code. The filter coefficients are stored in *FilterCoef.dat in ASCII form.*

```
./CLEAN           # Not necessary though cleans the CMAKE build file
./BUILD_AND_RUN   # Build and Run the CLSLP software
./GENERATE_PLOTS  # Generates GNUPLOT plot files (Amplitude and Phase)
```

4. FIR Filter Synthesis Examples

The functionality and use of the Constrained Least-Squares Linear-Phase CLSLP FIR Filter Synthesis algorithm is described via a series of simple examples in the following sections. These sections assume you have the software from the CLSLP Software Distribution described in Section 3.4.1 above.

4.1. Sloped-Bandpass Filter Example

We construct a linear-phase real-valued FIR filter to pass .1 to .4 Normalized Hz and null the other frequencies. The passband starts at .1 Hz with a -80 dB response. It rises either linearly or exponentially, depending on the type of test performed, to .4 Hz with a response of 0 dB. We use a 501-tap linear-phase real-valued FIR Filter and the CLSLP algorithm to perform the filter synthesis. The amplitude spectra are calculated by performing a Discrete Fourier Transform (DFT) on a zero-padded vector of filter coefficients.

All linear-phase FIR filters have conjugate-symmetric filter coefficients by the argument given in reference [1]. Since this is a real-valued filter, conjugate-symmetric corresponds to symmetric filter coefficients here.

4.1.1. Exponential/Absolute Weighted-Error Filter Segments

A linear-phase real-valued FIR filter with 501-coefficients is synthesized using the CLSLP algorithm. We are testing the *exponential/absolute* mode here. The synthesis takes *1572* µs or *3.14* µs per coefficient. The filter amplitude response is given in the graph below. The exponential filter segment appears as a straight line in the dB amplitude response below.
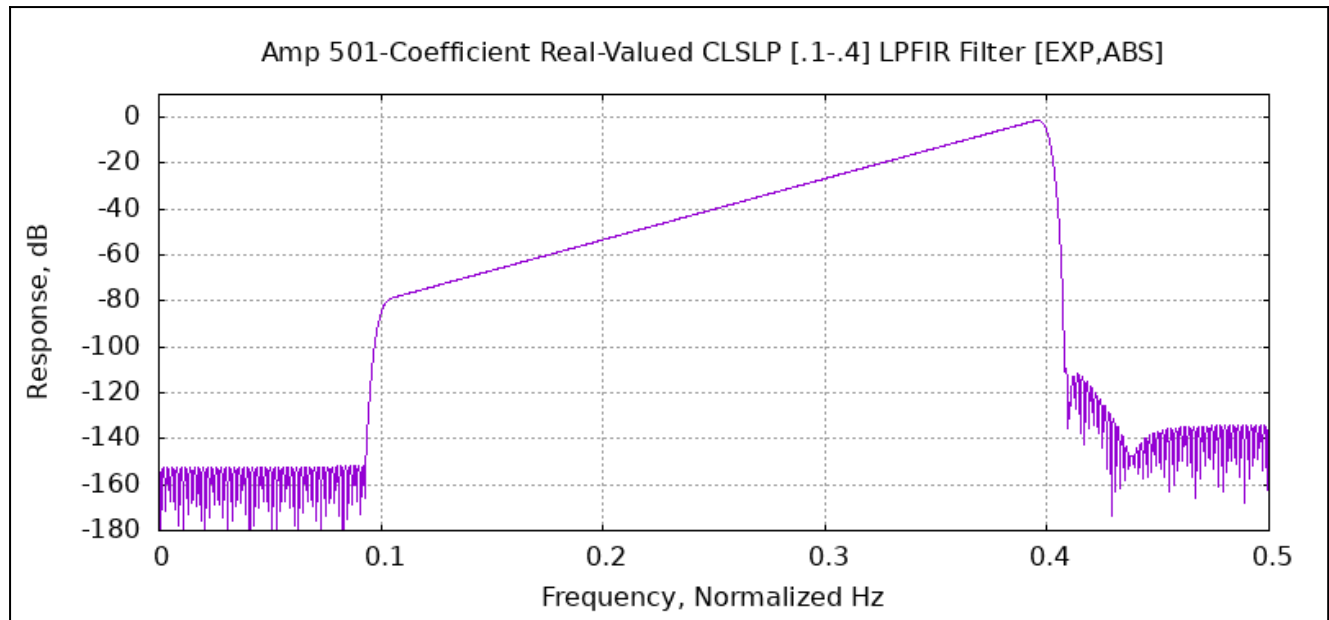

Figure 4.1: 501-Coefficient Real-Valued CLSLP [.1-.4] LPFIR Filter [EXP,ABS].

FIR filter synthesis is performed via the C++ code fragment below. The *AddSymmetric* function is used for real-valued filters [0-.5) and *Add* for complex-valued filters [0-1). *AddSymmetric* adds filter constraints that are conjugate-symmetric in frequency so as to guarantee real-valued filter coefficients. These coefficients are then written to the *OutCoefs* vector.

```
// The C++ code segment that generates this 501-coefficient CLSLP filter
#include "CLSLP.h"

// Generate the CLSLP filter Coefficients into OutCoefs
const unsigned int      NumFiltCoefs = 501;
CLSLP                   Filt;
vector<complex<double>>  OutCoefs(NumFiltCoefs);

// 'AddSymmetric' is for Real-Valued Filter Coefficients    ([0,.5] mod 1.) Normalized Hz
// 'Add'          is for Complex-Valued Filter Coefficients ([0,1.) mod 1.) Normalized Hz
Filt.AddSymmetric( LS_FIR_Filter::eLin_AbsErr, .0, .1, 0.00, 0.00, 1 );
Filt.AddSymmetric( LS_FIR_Filter::eExp_AbsErr, .1, .4, 1e-4, 1.00, 1 );
Filt.AddSymmetric( LS_FIR_Filter::eLin_AbsErr, .4, .5, 0.00, 0.00, 1 );

// Calculate the FIR Filter Coefficients into OutCoefs
Filt.GenFilter( NumFiltCoefs, OutCoefs );
```

4.1.2. Exponential/Relative Weighted-Error Frequency Intervals

A linear-phase real-valued FIR filter with 501-coefficients is synthesized using the CLSLP algorithm. We are testing the *exponential/relative* mode. This synthesis takes about *1556* µs or *3.11* µs per filter coefficient. The filter's amplitude response is given in Figure 4.2 below. The exponential filter interval appears as a sloped-line in the dB amplitude response. The little response peak just below .1 Hz can be reduced by adding a small triangular filter design section to reduce the abrupt amplitude transition. The filter response will improve as a result.
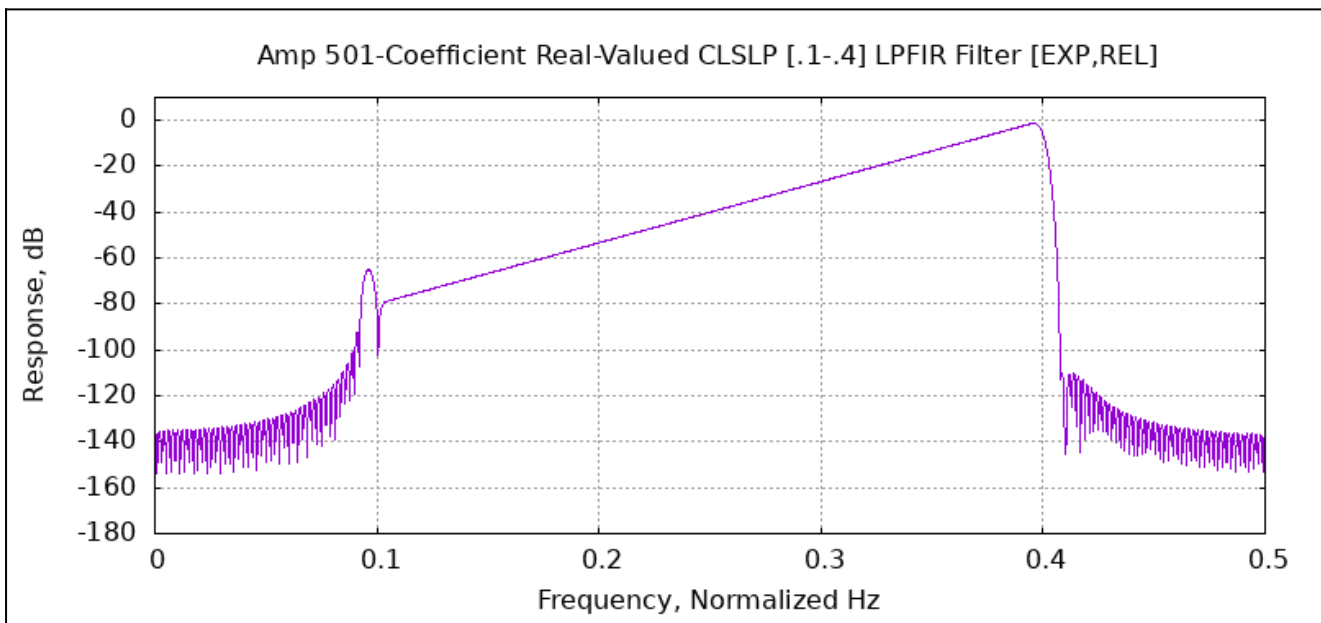

Figure 4.2: Amplitude: 501-Coefficient Real-Valued CLSLP [.1-.4] LPFIR Filter [EXP,REL].

The code fragment below is the C++ *main* program used in this example.

7

```
// The C++ code segment that generates this 501-coefficient CLSLP filter
#include "CLSLP.h"

// Generate the CLSLP filter Coefficients into OutCoefs
const unsigned int      NumFiltCoefs = 501;
CLSLP                   Filt;
vector<complex<double>>  OutCoefs(NumFiltCoefs);

// 'AddSymmetric' is for Real-Valued Filter Coefficients    ([0,.5] mod 1.) Normalized Hz
// 'Add'          is for Complex-Valued Filter Coefficients ([0,1.) mod 1.) Normalized Hz
Filt.AddSymmetric( LS_FIR_Filter::eLin_AbsErr, .0, .1, 0.00, 0.00, 1 );
Filt.AddSymmetric( LS_FIR_Filter::eExp_RelErr, .1, .4, 1e-4, 1.00, 1 );
Filt.AddSymmetric( LS_FIR_Filter::eLin_AbsErr, .4, .5, 0.00, 0.00, 1 );

// Calculate the FIR Filter Coefficients into OutCoefs
Filt.GenFilter( NumFiltCoefs, OutCoefs );
```

4.1.3. Linear/Absolute Weighted-Error Filter Segments

A linear-phase real-valued FIR filter with 501-coefficients is synthesized using the CLSLP algorithm. We are testing the *Linear/Absolute* mode. This 501-coefficient filter can be generated in *3517* µs or *7.02* µs per coefficient. The linear slope appears as a logarithmic curve.
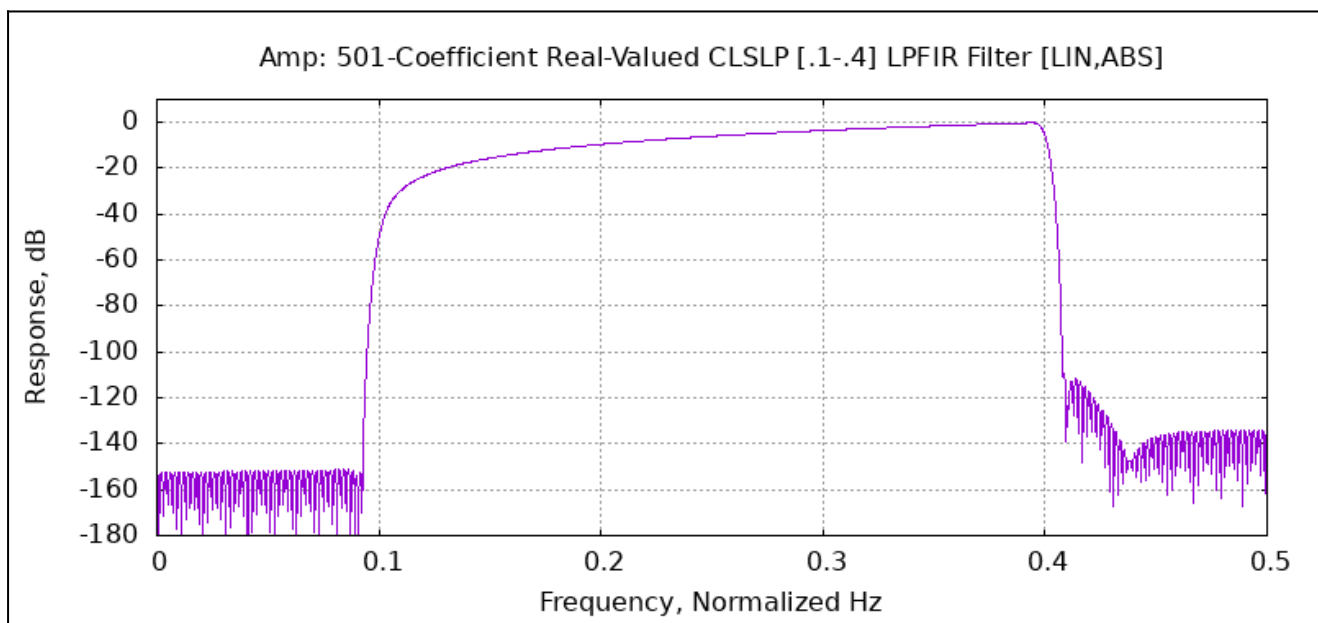

Figure 4.3: Amplitude: 501-Coefficient Real-Valued CLSLP [.1-.4] LPFIR Filter [LIN,ABS].

The code fragment below is the C++ *main* program used in this example.

8

```
// The C++ code segment that generates this 501-coefficient CLSLP filter
#include "CLSLP.h"

// Generate the CLSLP filter Coefficients into OutCoefs
const unsigned int      NumFiltCoefs = 501;
LS_FIR_Filter           Filt;
vector<complex<double>>  OutCoefs(NumFiltCoefs);

// 'AddSymmetric' is for Real-Valued Filter Coefficients    ([0,.5] mod 1.) Normalized Hz
// 'Add'          is for Complex-Valued Filter Coefficients ([0,1.) mod 1.) Normalized Hz
Filt.AddSymmetric( LS_FIR_Filter::eLin_AbsErr, .0, .1, 0.00, 0.00, 1 );
Filt.AddSymmetric( LS_FIR_Filter::eLin_AbsErr, .1, .4, 1e-4, 1.00, 1 );
Filt.AddSymmetric( LS_FIR_Filter::eLin_AbsErr, .4, .5, 0.00, 0.00, 1 );

// Calculate the FIR Filter Coefficients into OutCoefs
Filt.GenFilter( NumFiltCoefs, OutCoefs );
```

4.1.4. Linear/Relative Weighted-Error Filter Segments

A linear-phase FIR (LPFIR) filter with 501-coefficients is synthesized using the CLSLP algorithm. We are testing the *linear/relative* mode. The synthesis takes about *1635* µs or *3.26* µs per coefficient. The CLSLP filter amplitude response is given in Figure 4.4 below. The linear filter segment appears as a logarithmic curve in the amplitude response plot below. There is a small peak just below .1 Hz though we are close to making the -80 dB response. Adding a small triangular filter design section to reduce the abrupt amplitude transition. The filter response will improve substantially as a result. This 501-coefficient filter can be generated in *1635 µs* or *3.26 µs* per coefficient. The linear slope appears as a logarithmic curve.
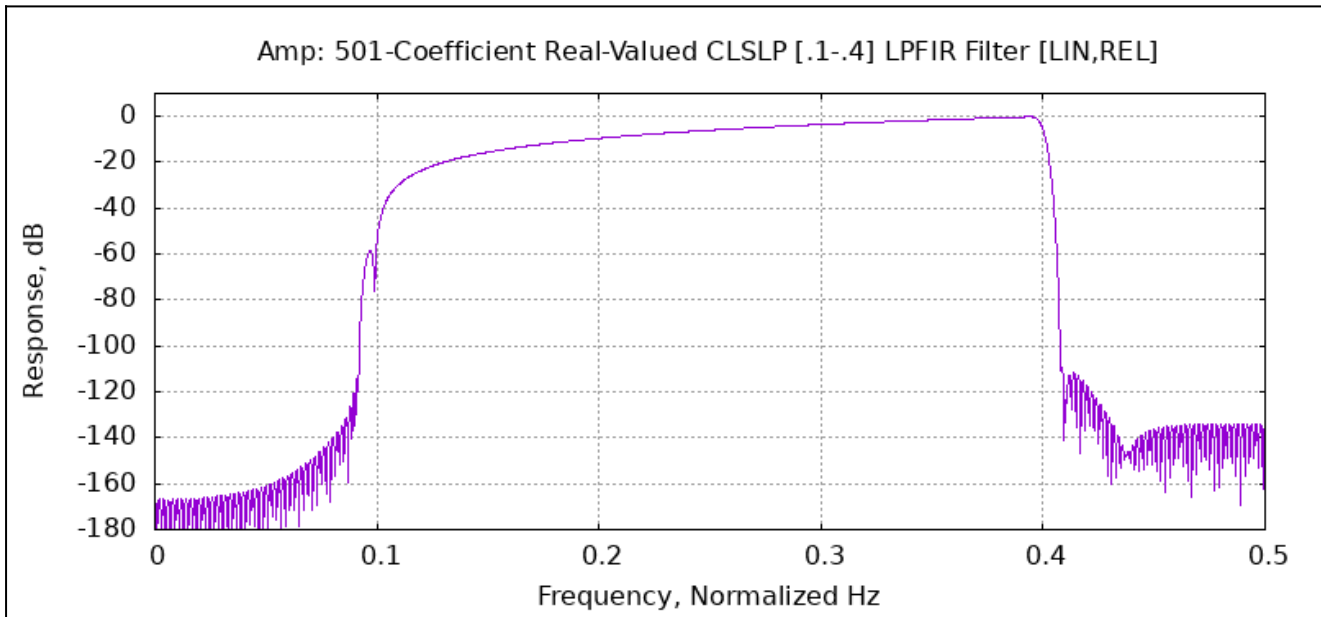

Figure 4.4: Amplitude: 501-Coefficient Real-Valued CLSLP [.1-.4] LPFIR Filter [LIN,REL].

The code fragment below is the C++ *main* program used in this example.

9

```
// The C++ code segment that generates this 501-coefficient CLSLP filter
#include "CLSLP.h"

// Generate the CLSLP filter Coefficients into OutCoefs
const unsigned int     NumFiltCoefs = 501;
LS_FIR_Filter          Filt;
vector<complex<double>> OutCoefs(NumFiltCoefs);

// 'AddSymmetric' is for Real-Valued Filter Coefficients    ([0,.5] mod 1.) Normalized Hz
// 'Add'          is for Complex-Valued Filter Coefficients ([0,1.) mod 1.) Normalized Hz
Filt.AddSymmetric( LS_FIR_Filter::eLin_AbsErr, .0, .1, 0.00, 0.00, 1 );
Filt.AddSymmetric( LS_FIR_Filter::eLin_RelErr, .1, .4, 1e-4, 1.00, 1 );
Filt.AddSymmetric( LS_FIR_Filter::eLin_AbsErr, .4, .5, 0.00, 0.00, 1 );

// Calculate the FIR Filter Coefficients into OutCoefs
Filt.GenFilter( NumFiltCoefs, OutCoefs );
```

4.2. Complex-Coefficient Linear-Phase FIR (LPFIR) Filters

The filter response of a real-valued LPFIR filter possess conjugate-symmetry about *0* Hz. Therefore it suffices to specify the the amplitude response only from *0* to *.5* Normalized Hz and infer the rest from the symmetry. FIR filters with complex-valued coefficients do not have conjugate symmetric filter coefficients. Therefore plots of complex-coefficient FIR filters are usually plotted from -.5 to .5 Normalized Hz.

4.2.1. Example Filter 5

A linear-phase FIR filter with 1001 complex-valued coefficients is synthesized using the CLSLP algorithm. We are testing the complex-coefficient mode so amplitude plots will be displayed from *-.5* to *.5* Normalized Hz.

```
// The C++ code segment that generates this 1001-coefficient CLSLP filter
#include "CLSLP.h"

// Generate the CLSLP filter Coefficients into OutCoefs
const unsigned int     NumFiltCoefs = 1001;
LS_FIR_Filter          Filt;
vector<complex<double>> OutCoefs(NumFiltCoefs);

// 'AddSymmetric' is for Real-Valued Filter Coefficients    ([0,.5] mod 1.) Normalized Hz
// 'Add'          is for Complex-Valued Filter Coefficients ([0,1.) mod 1.) Normalized Hz
Filt.Add( LS_FIR_Filter::eLin_AbsErr, .0,  .05,  0.00,  0.00,  1 );
Filt.Add( LS_FIR_Filter::eExp_AbsErr, .05,  .2,  1e-4,  1.00,  1 );
Filt.Add( LS_FIR_Filter::eExp_AbsErr, .2,   .5,  1e-4,  1.00,  1 );
Filt.Add( LS_FIR_Filter::eExp_AbsErr, .5,   .9,  1.00,  1e-4,  1 );
Filt.Add( LS_FIR_Filter::eLin_AbsErr, .9,  1.0,  0.00,  0.00,  1 );

// Calculate the FIR Filter Coefficients into OutCoefs
Filt.GenFilter( NumFiltCoefs, OutCoefs );
```

This filter synthesis takes *5544 µs* or *5.54 µs* per coefficient on a standard PC. The resultant CLSLP filter's amplitude response is given in Figure 3.2 below.
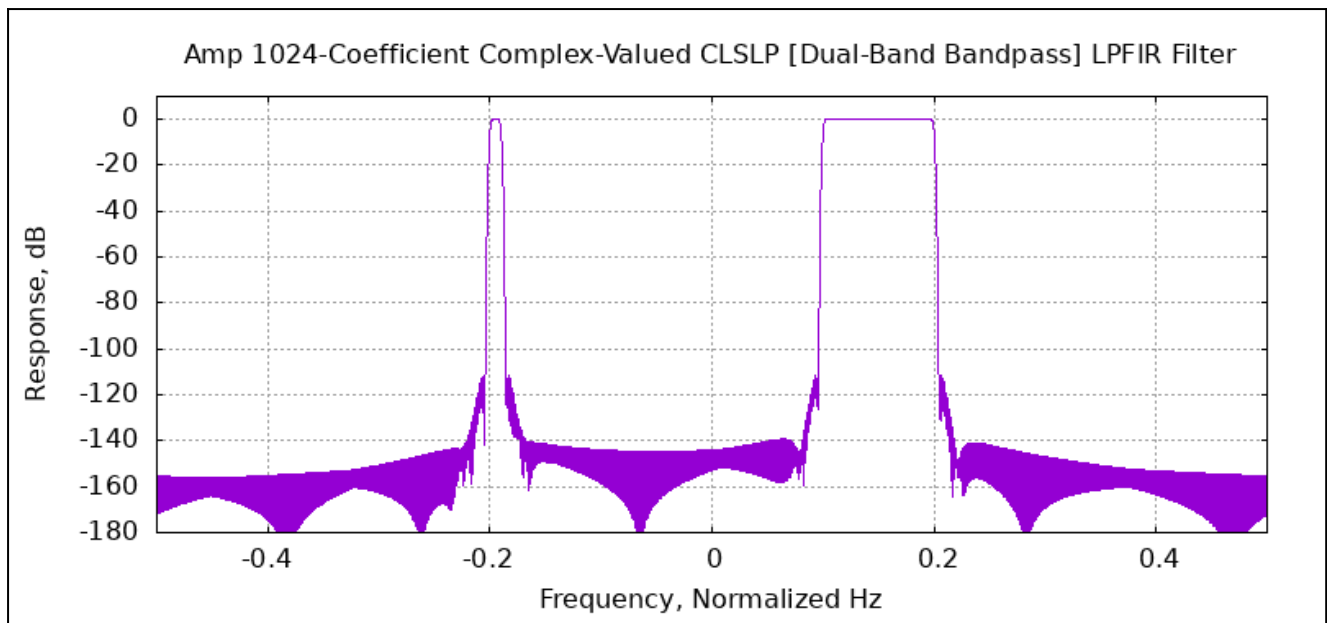
Figure 4.5: Amplitude: Ex5 1001-Coefficient Complex-Value CLSLP LPFIR [EXP,ABS].

4.2.2.   Example Filter 6

This filter synthesis takes *23388 µs* or *11.4255 µs* per coefficient on a standard PC.   The resultant CLSLP filter's amplitude response is given in Figure 3.2 below.

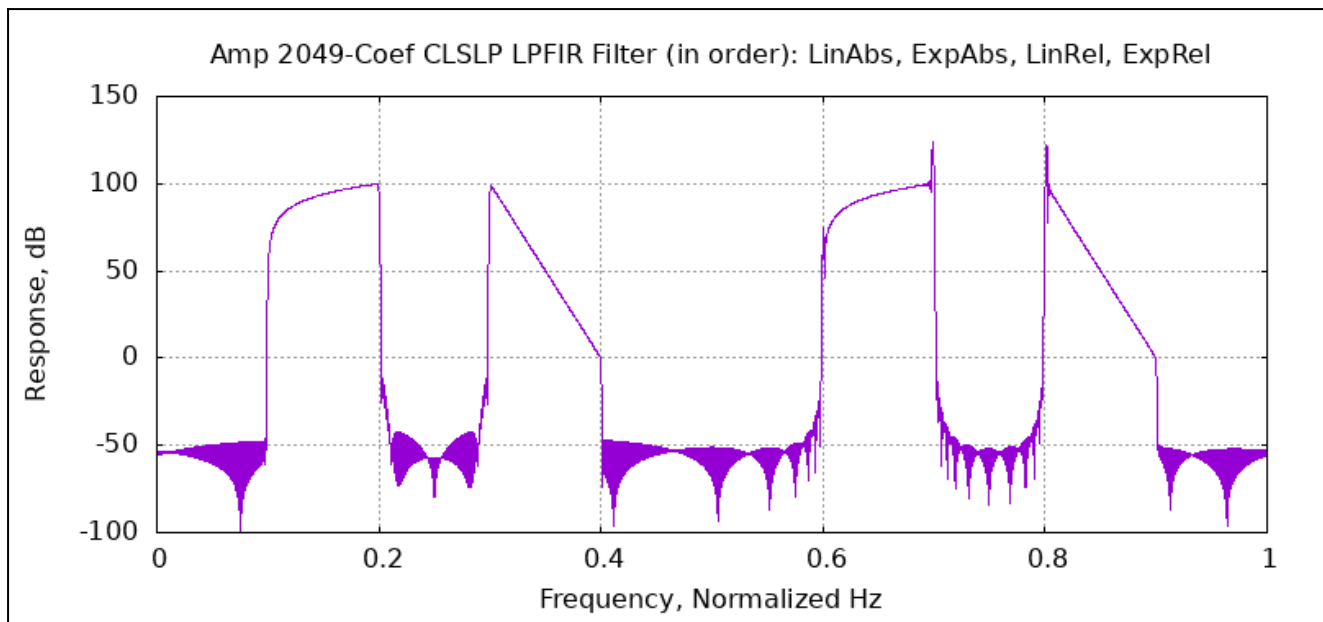*Figure 1: Amplitude: Ex6 2047-Coefficient Complex-Value CLSLP LPFIR [EXP,ABS].*

5. Web Resources

See website **http://www.wejc.com** for CLSLP software information and downloads. Look for the acronym CLSLP on the initial web page. The original Asilomar paper "*Constrained Least-Squares Design and Characterization of Affine Phase Complex FIR Filters*", reference [1], is included as PDF file, "*Constrained Least-Squares FIR Filter Synthesis.pdf*" which is included in the normal **CLSLP** software distribution.

The **http://www.wejc.com/** website has more information on other computer science, engineering, or mathematical algorithms as well as free downloads. If you have interest in any joint-work or just want to write, feel free to write at **wejc@wejc.com**. We are in *Washington State, United States*.

6. References

[1]   A. G. Jaffer and W. E. Jones, "Constrained least-squares design and characterization of affine phase complex FIR filters," Proceedings of 27th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, USA, 1993, pp. 685-691 vol.1, doi: 10.1109/ACSSC.1993.342607.

[2]   A. G. Jaffer and W. E. Jones, "Weighted least-squares design and characterization of complex FIR filters," in IEEE Transactions on Signal Processing, vol. 43, no. 10, pp. 2398-2401, Oct. 1995, doi: 10.1109/78.469851.

[3]   A. G. Jaffer, W. E. Jones and T. J. Abatzoglou, "Weighted least-squares design of linear-phase and arbitrary 2-D complex FIR filters," 1995 International Conference on Acoustics, Speech, and Signal Processing, Detroit, MI, USA, 1995, pp. 1256-1259 vol.2, doi: 10.1109/ICASSP.1995.480467.

CLSLP LPFIR Filter Synthesis Errata

Support Files Follow

Requires a Recent GNU or Equivalent C++ Compiler

[C++20 Language Standard Source Code Used]

Contents	
LS_FIR_Filter.h	The C++ header file for the filter synthesis software.
LS_FIR_Filter.cpp	The C++ source file for the filter synthesis software.
AlignedMemAlloc.h	SSE and AVX C++ vector allocator for faster execution on SIMD machines.
CLEAN	A BASH script that clears the CMAKE build directory.
BUILD_AND_RUN	A BASH script that performs a CMAKE build then runs the just-built program.
GENERATE_PLOTS	A BASH script that generates amplitude and phase plots, as *PNG* files, using *GNUPLOT*.
Start.cpp	The source code listing for the c++ *main* (where the program begins).
Constrained Least-Squares Design and Characterization of Affine Phase Complex FIR Filters	The Asilomar paper "*Constrained Least-Squares Design and Characterization of Affine Phase Complex FIR Filters*" ref [1]. Detail on the general theory behind CLSLP LPFIR filters with all the requisite math. The C++ subroutines used here for CLSLP, are coded directly from the definitions in this document.

Constrained Least-Squares Design and Characterization of Affine Phase Complex FIR Filters

Amin G. Jaffer and William E. Jones

Hughes Aircraft Company, Fullerton, CA

Abstract

In many signal processing applications, the need arises for the design of complex coefficient finite impulse response (FIR) filters to meet the specifications which cannot be approximated by real coefficient FIR filters. This paper presents a new technique for the design of complex FIR filters based on minimizing a weighted integral squared-error criterion subject to the constraint that the resulting filter response be affine phase (i.e., generalize linear phase). The technique makes use of the necessary and sufficient conditions for a causal complex FIR filter to possess affine phase which are explicitly derived here. The method is non-iterative and computationally efficient. Several illustrative filter design examples are presented with excellent results.

1. Introduction.

The subject of the design of finite impulse response filters has a long history, as evidenced by a partial list of publications [1]-[5]. For the most part, however, the previous publications have been concerned with the design of real coefficient FIR filters whose frequency response functions $H(f)$ necessarily satisfy $H(f)=H^*(-f)$, where * denotes complex conjugate. For a significant class of sensor signal processing problems, however, the desired frequency response will not necessarily satisfy this condition, e.g. the design of asymmetric notch filters for clutter cancellation problems in airborne radar or moving-platform active sonar systems. These systems and, in general, systems where analytic signals are to be processed to yield filter responses not satisfying this condition, mandate the need for complex coefficient FIR filters.

Although several authors have addressed the design of FIR filters by complex Chebyshev approximations [6]-[8], their works were restricted to real coefficient filters and their methods do not generalize readily to the complex coefficient case. Preuss [9] addressed the design of complex FIR filters using the Chebyshev norm and presented some interesting examples. However, his method involved a heuristic modification of the Remez exchange algorithm resulting in an iterative procedure that is not guaranteed to converge to the optimal solution. Weighted least-squares techniques [10], [11] also seem to have been applied only to the real coefficient case and, unlike the methods of this paper, appear to require a dense frequency sampling grid to model the desired amplitude response.

This paper is concerned with the design of complex coefficient FIR filters to satisfy a specified multiband amplitude response, based on minimizing a weighted integral squared-error criterion subject to the constraint that the resulting filter response possesses *affine* phase (i.e. linear phase with an offset). The incorporation of the affine phase constraint leads to good filter design and, moreover, is often a requirement in many system applications. The minimization is carried out subject to appropriate constraints on the filter coefficients (e.g. conjugate-symmetry constraints) needed to satisfy the affine phase property. These constraints for *complex* FIR filters are explicitly derived here in general form. An important feature of the present work is the use of the piecewise linear or exponential models to specify the desired multiband amplitude response, leading to an efficient, closed-form evaluation of certain integrals required in the computation of the optimal filter coefficient vector. This avoids the need for solving a discretized problem using a dense frequency sampling grid for the desired amplitude response, with the attendant problems in the transition bands. The filter design method requires only the solution of a set of N simultaneous linear equations (where N is the filter length) with a Hermitian-Toeplitz coefficient matrix, which can be obtained in only $o(N^2)$ computations using the efficient Levinson or Trench algorithms [13].

Some illustrative complex coefficient filter design examples are also presented, including the design of asymmetric notch filters required for clutter suppression, and bandpass differentiators, with excellent results in general and direct comparison to previously published results for the latter example. Further

examples, and the design of unconstrained complex FIR filters, can be found in [14].

2. Conditions for causal complex-valued FIR filters to possess affine phase

The conditions for real-valued causal FIR filters to possess linear phase, including linear phase with an offset, more appropriately termed *affine* phase, are well known and have been derived in [4] and [5]. However, their methods do not directly extend to the complex coefficient FIR filter case. It also appears that the necessary and sufficient conditions for complex FIR filters to be affine phase (including strict linear phase) do not appear to have been previously derived or stated, although special cases have been used in the literature [9]. The filter response for a causal N length FIR filter with complex coefficients $h(0), h(1), ..., h(N-1)$ is given by

$$H(f) = \sum_{n=0}^{N-1} h(n) e^{-j2\pi fn} \qquad (1)$$

where, in (1), the sampling interval is taken as equal to *1* second so that f represents normalized frequency. We deduce, in the following theorem, the necessary and sufficient conditions for $H(f)$ to have an affine phase function

$$\Phi(f) = -2\pi f\alpha + \beta \qquad (2)$$

for some α and β. The proof does not require separate treatments of even and odd length filters and is more general and precise than previous ones pertaining to the real filter case [5]. In order to exclude filters with leading or trailing zero coefficients which effectively alter the length of the filter, we impose the conditions $h(0) \neq 0$, $h(N-1) \neq 0$.

The main results of this Section are contained in the following theorem and its corollaries.

Theorem: *The filter response of a causal complex coefficient FIR filter, with coefficients $h(0), h(1), ..., h(N-1)$ where $h(0) \neq 0$, $h(N-1) \neq 0$, is affine phase of the form (2) if and only if*
$h(n) = e^{j2\beta} h^*(N-1-n)$, $\qquad n = 0, ..., N-1$
Furthermore, the delay term α is necessarily given by
$\alpha = (N-1)/2$

Proof: We prove necessity, i.e. the "only if" part, first. The affine phase property implies that (1) can be rewritten in the form

$$H(f) = e^{j\beta} e^{-j2\pi f\alpha} \left\{ e^{-j\beta} \sum_{n=0}^{N-1} h(n) e^{-j2\pi f(n-\alpha)} \right\} \qquad (3)$$

where the term inside the braces is purely real, i.e. equal to its conjugate. Hence, equating this term to its conjugate and letting $z \equiv e^{j2\pi f}$ results in, explicitly,

$$h(0)z^\alpha + h(1)z^{\alpha-1} + ... + h(N-1)z^{\alpha-N+1} \qquad (4)$$

$$= e^{j2\beta} \left[\begin{array}{c} h^*(0)z^{-\alpha} + h^*(1)z^{1-\alpha} + ... \\ + h^*(N-1)z^{N-1-\alpha} \end{array} \right]$$

In (4), z and its powers constitute a set of complex exponential functions that are linearly independent [15]. Hence (4) can only be satisfied for all values of z by equating the coefficients of like powers of z on both sides of (4). Since the powers of z on the left and right sides of (4) are, in descending order, $\{\alpha, \alpha-1, ..., \alpha-N+1\}$ and $\{N-1-\alpha, N-2-\alpha, ..., -\alpha\}$ respectively and since $h(0) \neq 0$, $h(N-1) \neq 0$ by assumption, we must have that the highest and lowest powers in the former set fall within the range of the latter set, i.e.,

$$-\alpha \leq \alpha \leq N-1-\alpha \qquad (5)$$
$$-\alpha \leq \alpha-N+1 \leq N-1-\alpha \qquad (6)$$

But then, it follows from (5) that $0 \leq \alpha \leq (N-1)/2$ and from (6) that $(N-1)/2 \leq \alpha \leq N-1$, from which it follows that

$$\alpha = (N-1)/2 \qquad (7)$$

precisely. Substituting (7) into (4), letting $m = N-1-n$ on the right side of (4) and rearranging yields

$$\sum_{n=0}^{N-1} h(n) z^{-(n-(N-1)/2)} = e^{j2\beta} \left\{ \sum_{n=0}^{N-1} h^*(N-1-n) z^{-(n-(N-1)/2)} \right\} \qquad (8)$$

Equating coefficients of same powers of z on both sides of (8) (because of linear independence of the complex exponentials of z and its powers) yields

$$h(n) = e^{j2\beta} h^*(N-1-n), \quad n = 0, ..., N-1 \qquad (9)$$

which, together with (7), proves the necessity part of the theorem. It is noted that setting $\beta = 0$ or $\pm \pi/2$ in (9) yields the conjugate symmetric and anti-conjugate symmetric filters respectively:

$$h(n) = h^*(N-1-n), \quad n = 0, ..., N-1 \qquad (10)$$

$$h(n) = -h^*(N-1-n), \quad n = 0, ..., N-1 \qquad (11)$$

Sufficiency: This follows immediately; for a proof see [14].

Corollary 1: If the filter coefficients are restricted to be real, it can be shown that the conditions (9) result in just two different constraints
$$h(n) = h(N-1-n), \qquad n = 0, ..., N-1 \qquad (12)$$

$$h(n) = -h(N-1-n), \qquad n = 0,\ldots,N-1 \qquad (13)$$

which define the symmetric and anti-symmetric filters respectively, in agreement with [5]. The proof is omitted here for lack of space but can be found in [14].

Corollary 2: We will show that it suffices for filter design applications to apply the theorem for $\beta = 0$ and rotate the output phase by β. In (3), let $h_0(n) = e^{-j\beta} h(n)$ where the $h(n)$ satisfy the conditions of the theorem. For $n = 0,\ldots,N-1$, we have

$$h(n) = e^{j2\beta} h^*(N-1-n) \qquad (14)$$

and

$$h_0(n) = e^{j\beta} h^*(N-1-n) \qquad (15)$$

$$= h_0^*(N-1-n)$$

Hence $h_0(n)$ satisfies the conditions of the theorem for strict linear phase (with $\beta = 0$). Hence the filter response (3) can be written as

$$H(f) = e^{j\beta} H_0(f) \qquad (16)$$

where

$$H_0(f) = e^{-j2\pi a} \left\{ \sum_{n=0}^{N-1} h_0(n) e^{-j2\pi f(n-a)} \right\} \qquad (17)$$

with $a = (N-1)/2$. $H_0(f)$, as given by (17), is the frequency response of a strict linear phase filter and hence (16) states that the frequency response of an arbitrary affine phase filter can be obtained by multiplying $H_0(f)$ by $e^{j\beta}$, as was to be shown.

3. Linear phase FIR filter design by constrained weighted least-squares method.

We consider here the problem of designing complex coefficient FIR filters to approximate a specified complex valued frequency response by employing a weighted integral least-squares error criterion. Let $z_D(f)$ be the desired complex valued frequency response and $W(f)$ be a real nonnegative piecewise continuous weighting function. $z_D(f)$ is of the form $z_D(f) = a_D(f) e^{j\Phi_D(f)}$ where $a_D(f)$ and $\Phi_D(f)$ are the desired amplitude and phase responses respectively. Let $\underline{h} = [\, h(0), h(1), \ldots, h(N-1) \,]^T$ represent the complex coefficient FIR filter of length N. $H(f)$, as given by (1) can be written as

$$H(f) = \underline{d}^H(f)\underline{h} \qquad (18)$$

$$\textit{where} \qquad \underline{d}(f) = \left[\, 1, e^{j2\pi f}, \ldots, e^{j2\pi(N-1)f} \,\right]^T$$

and the superscripts T and H denote the transpose and conjugate transpose operators. We seek to obtain the coefficient vector $\underline{h}$ that minimizes the weighted integral squared-error criterion

$$J(\underline{h}) = \int_0^1 W(f) \left| z_D(f) - \underline{d}^H(f)\underline{h} \right|^2 df \qquad (19)$$

subject to the constraint (1). It is noted that the criterion (19) allows for $z_D(f)$ being specified over compact subsets of the normalized frequency interval $[0,1)$ and that $W(f)$ may equal zero over some of the subsets. In the design examples given in Section 5, $W(f)$ is chosen so that criterion (19) becomes one of minimizing the *relative* integral squared error. This is discussed more fully in Section 4.

The conjugate symmetry constraint (10) on the filter coefficients can be compactly represented as

$$\underline{h}^* = E\underline{h} \qquad (20)$$

where E is the $N \times N$ exchange matrix with ones on the cross-diagonal and zeros elsewhere. The exchange matrix has the properties that $E^T = E$ and $E^2 = I$ (the identity matrix) so that $E^{-1} = E$. Note also that the formulation (20) applies to both even and odd length filters, so that separate treatments of these two cases are unnecessary.

The cost function (19) is a real-valued, nonnegative function of the complex vector $\underline{h}$. Its minimization may be accomplished by the use of a complex gradient operator and its associated matrix-vector calculus operations as described by Brandwood [13]. Let $\underline{h} = \underline{h}_x + j\underline{h}_y$ where $\underline{h}_x$ and $\underline{h}_y$ are the real and imaginary components of the complex vector $\underline{h}$. Define the complex gradient operator as

$$\nabla_{\underline{h}} = 1/2 \left(\partial/\partial\underline{h}_x - j\,\partial/\partial\underline{h}_y \right) \qquad (21)$$

Then a necessary and sufficient condition for a stationary point of $J(\underline{h})$ is that $\nabla_{\underline{h}} J(\underline{h}) = \underline{0}$. Equation (19) can be written as

$$J(\underline{h}) = \int_0^1 W(f) \left[\begin{array}{c} |z_D(f)|^2 - z_D^*(f)\underline{d}^H(f)\underline{h} \\ -\underline{h}^H \underline{d}(f)z_D(f) + \underline{h}^H \underline{d}(f)\underline{d}^H(f)\underline{h} \end{array} \right] df \qquad (22)$$

Using the constraint equation $\underline{h}^* = E\underline{h}$ or, equivalently, $\underline{h}^H = \underline{h}^T E$, we get

$$J(\underline{h}) = \int_0^1 W(f) \left[\begin{array}{c} |z_D(f)|^2 - z_D^*(f)\underline{d}^H(f)\underline{h} \\ -\underline{h}^T E\underline{d}(f)z_D(f) \\ +\underline{h}^T E\underline{d}(f)\underline{d}^H(f)\underline{h} \end{array} \right] df \qquad (23)$$

Differentiating with respect to $\underline{h}$ (see [13] for details on applying the complex gradient operator to linear and quadratic forms), and equating to $\underline{0}$ yields

$$\nabla_{\underline{h}} J(\underline{h}) = \int_0^1 W(f) \left[\begin{array}{c} -z_D^*(f)\underline{d}^*(f) - E\underline{d}(f)z_D(f) \\ +\left(R + R^T\right)\underline{h} \end{array} \right] df = \underline{0} \qquad (24)$$

where $R = E\,\underline{d}(f)\,\underline{d}^H(f)$. Rearrangement of (24) yields

$$\left[\int_0^1 W(f)\left(R + R^T\right) df\right]\underline{h} = \tag{25}$$

$$\int_0^1 W(f)\left[E\,z_D(f)\underline{d}(f) + \overset{*}{z}_D(f)\underline{d}^*(f)\right] df$$

Although (25) expresses the filter coefficient vector $\underline{h}$ as the solution of a set of simultaneous linear equations, it can be simplified further to yield a more compact and computationally efficient form. First note that since $R = E\,\underline{d}(f)\,\underline{d}^H(f)$ and pre-multiplication of a matrix by E reverses its rows, we can readily show that $R = R^T$. Now let

$$Q = \int_0^1 W(f)\,\underline{d}(f)\,\underline{d}^H(f)\,df \tag{26}$$

$$\underline{r} = \int_0^1 W(f)\left[E\,z_D(f)\,\underline{d}(f) + z_D^*(f)\,\underline{d}^*(f)\right] df \tag{27}$$

Then (25) reduces to

$$Q\,\underline{h} = 1/2\,E\,\underline{r} \tag{28}$$

The Hermitian matrix Q is also Toeplitz since its $(m,n)th$ element, given by $\int_0^1 w(f)\,e^{j2\pi(m-n)f}\,df$, depends only on the difference $(m-n)$. Equation (28) represents a system of N simultaneous linear equations, with a Hermitian-Toeplitz coefficient matrix, for the solution of the filter coefficient vector $\underline{h}$. Additionally, since the Vandermonde type vectors $\underline{d}(f)$ are linearly independent for distinct values of f in the interval $[0,1)$, the matrix Q will be of full rank N (and positive-definite) as long as the range of the integration in (28) encompasses any interval or at least N discrete distinct points in the frequency domain where $W(f)$ is not zero. This will be true for any non-trivial, well-posed filter design problem, resulting in a unique solution for $\underline{h}$. The matrix Q is completely specified by either its first row or column and the system of linear equations (28) can be efficiently solved in $o(N^2)$ operations by the Levinson recursion or Trench algorithm [12], resulting in a significant computational savings over general matrix inversion techniques which require $o(N^3)$ operations.

Note that $\underline{h}$, as given by the solution of (28) actually satisfies the conjugate-symmetry constraint regardless of the specification of the desired phase response $\Phi_D(f)$, which need not be linear. However, since the constraint (20) would only be imposed for linear phase filter design problems, it would be appropriate for the desired phase response $\Phi_D(f)$ to be specified as linear phase with a delay of $(N-1)/2$, i.e. as $\Phi_D(f) = -2\pi f(N-1)/2$.

The Q matrix and the $\underline{r}$ vector, required in (28) are specified as follows: The first column of the Hermitian-Toeplitz matrix Q, which completely defines Q, is given by

$$[Q]_{m,1} = \int_0^1 W(f)\,e^{j2\pi(m-1)f}\,df, \qquad m = 1,\dots,N \tag{29}$$

Using $\Phi_D(f) = -2\pi f(N-1)/2$, the nth element of the $\underline{r}$ vector specified by (27) can be shown to be given by

$$[\underline{r}]_n = 2\int_0^1 W(f)\,a_D(f)\,e^{-j2\pi f(n-(N-1)/2)}\,df \tag{30}$$

In the filter design examples presented in Section 5 the weighting function corresponds to the squared *relative* error. Furthermore, since most filter design problems, including the examples in Section 5, require approximating the frequency response over multiple subbands, which may be disjoint, the weighting function specializes to

$$W(f) = \frac{c_k}{a_{Dk}^2(f)}, \qquad \begin{array}{l} F_{k1} \le f \le F_{k2} \\ k = 1,\dots,M \end{array} \tag{31}$$

where M is the number of frequency subbands of interest, $a_{Dk}(f)$ is the desired amplitude response over the kth subband and c_k are additional discrete weights included in (31) to permit emphasizing certain frequency segments over others. Under these assumptions, (29) and (30) reduce to

$$[Q]_{m,1} = \sum_{k=1}^M c_k \int_{F_{k1}}^{F_{k2}} \frac{e^{j2\pi f(m-1)}}{a_{Dk}^2(f)}\,df, \qquad m = 1,\dots,N \tag{32}$$

$$[\underline{r}]_n = 2\sum_{k=1}^M c_k \int_{F_{k1}}^{F_{k2}} \frac{e^{-j2\pi f(n-(N-1)/2)}}{a_{Dk}(f)}\,df, \qquad n = 1,\dots,N \tag{33}$$

Two particular types of amplitude response functions $a_{Dk}(f)$, the linear amplitude and the linear log-amplitude or exponential response models, are employed in the examples of Section 5. These result in the closed-form expressions for the integrals in (32) and (33) which are readily evaluated and also yield excellent performance as demonstrated in Section 5.

4. Weighting and amplitude response models.

In this section two specific amplitude response functions, the linear and exponential, used in conjunction with relative square-error weighting, are used as models for the linear phase complex FIR filter design technique, the constrained technique, defined in Section 4. The linear model is defined by

$$a_{Dk}(f) = \alpha_k + \beta_k\,f \tag{34}$$

$$\textit{where} \qquad \beta_k = \left(\frac{A_{k2} - A_{k1}}{F_{k2} - F_{k1}}\right)$$

$$\textit{and} \qquad \alpha_k = A_{k1} - F_{k1}\,\beta_k$$

and the exponential model by

$$a_{Dk}(f) = e^{\gamma_k + \delta_k f} \tag{35}$$

$$where \quad \delta_k = \left(\frac{\ln(A_{k2}) - \ln(A_{k1})}{F_{k2} - F_{k1}}\right)$$

$$and \quad \gamma_k = \ln(A_{k1}) - F_{k1}\delta_k$$

for the kth frequency segment.

The exponential model implies that the log-amplitude is linear over the frequency segment which is particularly suitable for many applications. The exponential model is also computationally more efficient than the linear model when they are used in conjunction with relative squared-error weighting. The Q matrix and $\underline{r}$ vector used in the constrained algorithm described in Section 3 are evaluated for the exponential model substituting (35) into the kth term of (32) and (33) to yield the expressions

$$[Q_k]_{m,1} = c_k \int_{F_{k1}}^{F_{k2}} \frac{e^{j2\pi f(m-1)}}{e^{2(\gamma_k + \delta_k f)}} df, \quad m = 1,\ldots,N \tag{36}$$

$$[\underline{r}_k]_n = 2c_k \int_{F_{k1}}^{F_{k2}} \frac{e^{-j2\pi f(n-(N-1)/2)}}{e^{(\gamma_k + \delta_k f)}} df, \quad n = 1,\ldots,N \tag{37}$$

These expressions can then be solved in closed form as

$$[Q_k]_{m,1} = c_k e^{-2\gamma_k} \frac{e^{2f(j\pi(m-1)-\delta_k)}}{2(j\pi(m-1)-\delta_k)}\bigg|_{f=F_{k1}}^{f=F_{k2}} \quad m = 1,\ldots,N \tag{38}$$

$$[\underline{r}_k]_n = -2c_k e^{-\gamma_k} \frac{e^{-f(j2\pi(n-(N-1)/2)+\delta_k)}}{(j2\pi(n-(N-1)/2)+\delta_k)}\bigg|_{f=F_{k1}}^{f=F_{k2}} \quad n = 1,\ldots,N \tag{39}$$

Note that (36) and (37) reduce to simpler expressions when $m = 1$ and $\delta_k = 0$ for (36), and when $n = (N+1)/2$ and $\delta_k = 0$ for (37).

The linear segment model used in conjunction with the relative squared-error weighting, for the constrained algorithm is determined by the Q matrix and $\underline{r}$ vector. These are formed by substituting (34) into (32) and (33) yielding the expressions

$$[Q_k]_{m,1} = c_k \int_{F_{k1}}^{F_{k2}} \frac{e^{j2\pi f(m-1)}}{(\alpha_k + \beta_k f)^2} df, \quad m = 1,\ldots,N \tag{40}$$

$$[\underline{r}_k]_n = 2c_k \int_{F_{k1}}^{F_{k2}} \frac{e^{-j2\pi f(n-(N-1)/2)}}{(\alpha_k + \beta_k f)} df, \quad n = 1,\ldots,N \tag{41}$$

The general result of evaluating (40) and (41) is given by

$$[Q_k]_{m,1} = \frac{c_k}{\beta_k} e^{-j2\pi(m-1)\frac{\alpha_k}{\beta_k}} \tag{42}$$

$$\bullet \left[\left(j2\pi\frac{(m-1)}{\beta_k}\right) Ei\left(j2\pi(m-1)\frac{u}{\beta_k}\right) - \frac{1}{u}e^{j2\pi(m-1)\frac{u}{\beta_k}}\right]_{u=A_{k1}}^{u=A_{k2}}$$

$$[\underline{r}_k]_n = \frac{2c_k}{\beta_k} e^{j2\pi(n-(N-1)/2)\frac{\alpha_k}{\beta_k}} \tag{43}$$

$$\left[Ei\left(-j2\pi(n-(N-1)/2)\frac{u}{\beta_k}\right)\right]_{u=A_{k1}}^{u=A_{k2}}$$

where $Ei(x)$ is the exponential integral as defined in Gradshteyn and Ryzhik [16] and Abramowitz and Stegun [17]. It can be easily evaluated in terms of the sine and cosine integrals $si(x)$ and $ci(x)$ related by

$$Ei(\pm jx) = ci(x) \pm j\, si(x) \tag{44}$$

Note that (40) and (41) reduce to simpler expressions when $\beta_k = 0$ or $m = 1$ for (40), and when $\beta_k = 0$ or $n = (N+1)/2$ for (41).

5. Illustrative filter design examples.

This section examines two applications of the constrained algorithm, the asymmetric v-notch filter and the bandpass differentiator examined by Preuss [9]. The examples in this section employ the amplitude response models of Section 4 used in conjunction with relative square-error weighting.

The filter design examples in this Section were generated using a MATLAB program based on the results of Sections 3 and 4. By exploiting the Toeplitz structure of matrix Q in as defined in Section 3, the Levinson or Trench algorithms [12] can be employed to reduce the computational complexity of the solution from an $o(n^3)$ solution to an $o(n^2)$ solution. Using these techniques the 101-tap V-Notch filter design problem below was generated in 3.35 seconds in MATLAB on an 486DX IBM-PC clone running at 50 MHz.

The **Asymmetric V-Notch Filter Design Problem:** The asymmetric V-Notch filter examined here is defined by the specification

$$|H_D(f)| = \begin{cases} 0 \; dB, & 0 \le f < .5 \\ [0,-40] \; dB, & .5 \le f < .7 \\ [-40,0] \; dB, & .7 \le f < .8 \\ 0 \; dB, & .8 \le f < 1.0 \end{cases} \tag{45}$$

where the quantities in brackets in (45) are the end-points of the *exponential* amplitude response function over the frequency interval specified ($[A_{k1},A_{k2}]$ for the kth frequency interval in (34)). The asymmetry in the desired amplitude response can arise in moving platform radar or active sonar systems where the unwanted clutter returns exhibit Doppler shifts that are largely "down-Doppler" relative to the Doppler frequency of the moving platform. Due to this asymmetry, the filter cannot be synthesized by conventional real FIR filter design techniques as the

coefficients can never be represented as a purely real sequence. For this example, each frequency subband is equally weighted in the sense that the c_k parameter defined in (31) is equal to *1* for each segment. Equations (28), (38) and (39) are used to evaluate the filter coefficients as functions of the parameters specified in the design specification.

Figure 1 illustrates a sample asymmetric v-notch filter generated with 101 coefficients. The filter coefficients are indeed conjugate-symmetric as expected from the arguments made in Section 2. The relative error of the filter synthesis

$$e_{rel,dB}(f) = 20.*\log\frac{|H_a(f)|}{|H_d(f)|} \qquad (46)$$

is plotted in Figure 2. The RMS error used here is defined by

$$RMS = \sqrt{\sum_{m=1}^{M} \int_{F_{m1}}^{F_{m2}} \left|\frac{|H_a(f)|-|H_d(f)|}{|H_d(f)|}\right|^2 df} \qquad (47)$$

The 101-tap linear phase V-notch filter achieves maximum relative error of about 0.47 dB and an rms error of .004759.
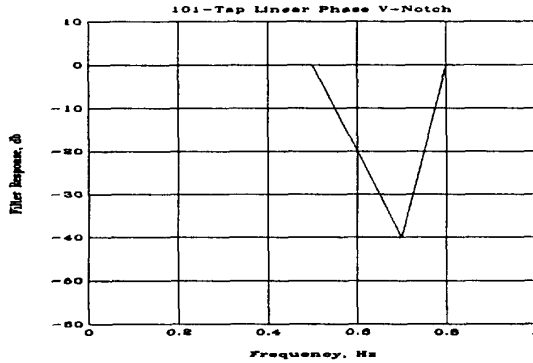

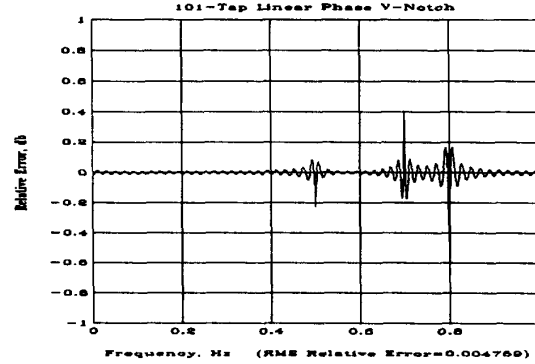Figure 1. 101-Tap Asymmetric V-Notch Filter Amplitude Response.

Figure 2. 101-Tap Asymmetric V-Notch Filter Error.

The Bandpass Differentiator: In this Section the linear phase bandpass differentiator filter specified by Preuss [9] is generated using the constrained technique as described in Sections 3 and 4. In Preuss' paper this filter is specified as

$$H_d(f) = \begin{cases} j2\pi f \, e^{-j2\pi 15.5 f} & 0.03750 \le f \le 0.42500 \\ 0 & 0.57500 \le f \le 0.96250 \end{cases} \qquad (48)$$

Note that this filter contains an affine phase offset produced by the *j* constant in (48). By the argument given in Section 2, it suffices to ignore this affine phase offset for the purposes of generating a linear phase filter and later multiply the complex FIR filter coefficients by this *j* term which forms the final filter. This post-multiplication doesn't change the filter's amplitude response.

The realization of the bandpass differentiator as specified in (48) using the constrained algorithm is given by

$$|H_d(f)| = \begin{cases} [.0710,.8700] & c_k = 2 \times 10^6, & .0355 \le f \prec .4350 \\ [.8700,.0009] & c_k = 100, & .4350 \le f \prec .5650 \\ [.0009,.0009] & c_k = 1, & .5650 \le f \prec .9625 \end{cases}$$

where the quantities in brackets are the end-points of the *linear* amplitude response function over the frequency interval specified ($[A_{k1},A_{k2}]$ for the *k*th frequency interval in (35)). The quantities c_k are the auxiliary weights applied to the subbands as defined in (31). This realization was found to yield satisfactory fit errors in the passband while preserving acceptable stop-band rejection. It is also desirable to weight the stop-band to a lesser degree than the passband to prevent an inordinate amount of effort being employed in flattening the stop-band. The results of our realization of the bandpass differentiator can be seen in Figure 3.

The filter design fit error used in this example is specified by

$$e_{rel}(f) = \frac{|H_a(f)| - |H_d(f)|}{|H_d(f)|} \qquad (49)$$

Preuss obtained a relative peak amplitude error of $\pm 2 \times 10^{-4}$ over the frequency interval $[.0375, .4250]$. The results of the application of the constrained algorithm are given in Figures 3 and 4. Note that the constrained algorithm achieves a smaller relative error over most of the passband (rms value of 1.084×10^{-4}), although the peak relative error is greater at about -4.3×10^{-4}. This result is consistent with the nature of the weighted integral squared error and the Chebyshev criteria.
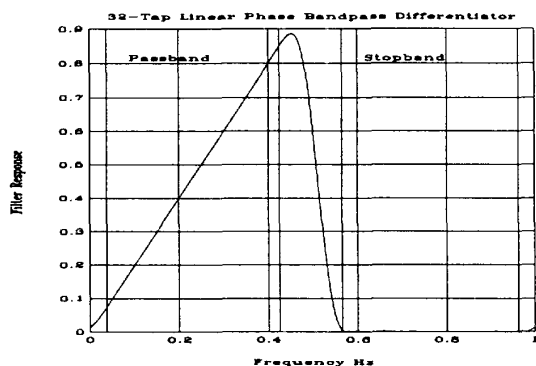
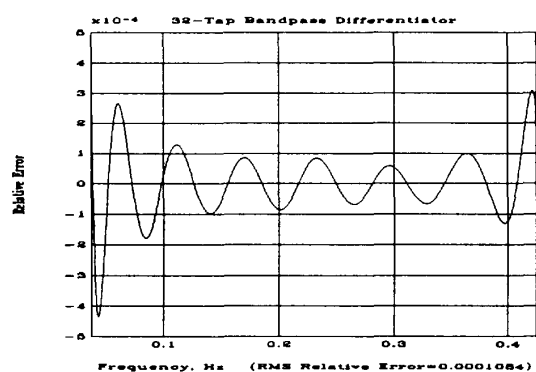

Fig 3. 32-Tap Bandpass Differentiator Amplitude Response.

Fig 4. 32-Tap Bandpass Differentiator Filter Fit Error.

References

[1]   L. R. Rabiner and R. W. Schafer, "Recursive and non-recursive realizations of digital filters designed by frequency sampling techniques," IEEE Trans. Audio Electroacoustics, vol. AU-19, no. 3, pp. 200-207, Sept. 1971.

[2]   T. W. Parks and J. H. McClellan, "Chebyshev approximations for non-recursive digital filters with linear phase," IEEE Trans. Circuit Theory vol. CT-19, no. 2, pp. 189-194, March 1972.

[3]   J. H. McClellan and Parks, "A unified approach to the design of optimal FIR linear phase digital filters," IEEE Trans. Circuit Theory, vol. CT-20, pp. 697-701, Nov. 1973.

[4]   L. R. Rabiner and B. Gold, Theory and Applications of Digital Signal Processing, Prentice-Hall, Inc., New Jersey, 1975.

[5]   A. V. Oppenheim and R. W. Schafer, Discrete-Time Signal Processing. Edgewood Cliffs, New Jersey, Prentice-Hall, 1989.

[6]   X. Chen and T. W. Parks, "Design of FIR filters in the complex domain," IEEE Trans. Acoust., Speech, Signal Processing, vol. ASSP-35, no. 2, pp. 144-153, Feb. 1987.

[7]   N. N. Chit and J. S. Mason, "Complex Chebyshev approximation for FIR digital filters," IEEE Trans. Signal Processing, vol. 39, No. 1, pp. 49-54, Jan. 1991.

[8]   A. S. Alkhairy, K. G. Christian and J. S. Lim, "Design and characterization of optimal FIR filters with arbitrary phase," IEEE Trans. Signal Processing, vol. 41, No. 2, pp. 559-572, Feb. 1993.

[9]   K. Preuss, "On the design of FIR filters by complex Chebyshev approximation," IEEE Trans. Acoust., Speech and Signal Processing, vol. 37, No. 5, pp. 702-712, May 1989.

[10]   Y. C. Lim, J. H. Lee, C. K. Chen and R. H. Yang, "A weighted least squares algorithm for quasi-equiripple FIR and IIR digital filter design," IEEE Trans. Signal Processing, vol. 40, No. 3, pp. 551-558, March 1992.

[11]   C. S. Burrus, A. W. Soewito and R. A. Gopinath, "Least squared error FIR filter design with transition bands," IEEE Trans. Signal Processing, vol. 40, No. 6, pp. 1327-1340, June 1992.

[12]   S. L. Marple, Digital Spectral Analysis with Applications. Englewood Cliffs, New Jersey, Prentice-Hall, 1987.

[13]   D. H. Brandwood, "A complex gradient operator and its applications in adaptive array theory," IEE Proceedings, vol. 130, Pts. F and H, No. 1, pp. 11-16, February 1983.

[14]   A. G. Jaffer and W. E. Jones, "Weighted Least-Squares Design and Characterization of Complex FIR Filters", submitted to IEEE Trans. Signal Processing.

[15]   Tom M. Apostol, Calculus, Volume II. Second edition, John Wiley & Sons, New York, 1969.

[16]   I. S. Gradshteyn and I. M. Ryzhik, Table of Integrals, Series, and Products. Orlando, Florida: Academic Press, 1980.

[17]   M. Abramowitz and I. E. Stegun, Handbook of Mathematical Functions. Washington, D.C.: National Bureau Of Standards Applied Mathematics Series #55, 1972.

Correspondence

Weighted Least-Squares Design and Characterization of Complex FIR Filters

Amin G. Jaffer and William E. Jones

Abstract— This correspondence presents two novel weighted least-squares methods for the design of complex coefficient finite impulse response (FIR) filters to attain specified arbitrary multiband magnitude and linear or arbitrary phase responses. These methods are computationally efficient, requiring only the solution of a Toeplitz system of N linear equations for an N-length filter that can be obtained in $o(N^2)$ operations. Illustrative filter design examples are presented.

I. INTRODUCTION

The subject of real FIR filter design using both the weighted least-squares error (WLS) and Chebyshev criteria has been addressed extensively in the past [1]–[4]. More recently, the design of complex FIR filters that satisfy specified asymmetric amplitude or phase responses necessary in radar/sonar clutter suppression problems and other applications has been considered [5]–[9]. Nguyen [7] and Pei and Shyu [8] have employed the eigenfilter technique to approximately optimize the complex FIR filter WLS error design criterion. The eigenfilter technique, in addition to being only approximately optimal, requires the computation of a principal eigenvector by an iterative technique, where the number of iterations required for convergence can be quite large, resulting in heavy computational demands.

Two complex FIR filter WLS synthesis techniques—one for arbitrary phase response (unconstrained method) and the other incorporating the linear phase constraint (constrained method)—are developed here. The direct WLS optimization methods presented here utilize the complex gradient operator [10], which avoids decomposing the complex variables into real and imaginary parts. The linear-phase constrained method is developed using the complex Lagrange multiplier constraint, which is valid for either odd or even length filters. The filter coefficient vector is obtained very efficiently for both techniques as the solution of the resulting Hermitian–Toeplitz system of linear equations using a noniterative method (Levinson algorithm) [11]. Additionally, for a special but useful class of filters, our techniques result in a solution that altogether avoids the need for matrix inversion or the solution of a system of linear equations, thus reducing the computational demands significantly. The relationship between the constrained and unconstrained techniques is also examined. Finally, two illustrative filter design examples are presented with direct comparison of example two with the eigenfilter design example of Nguyen [7].

II. WEIGHTED LEAST-SQUARES COMPLEX FIR FILTER DESIGN

We derive here weighted least-squares algorithms for designing complex FIR filters to approximate arbitrary magnitude response

constrained to have affine (generalized linear) phase as well as FIR filters with arbitrarily specified magnitude and phase responses. The conditions for complex FIR filters to posses affine phase are known in the literature and are also explicitly derived in [9]. Although the affine phase conditions are slightly more general, it suffices for our purposes to incorporate only the conjugate-symmetric constraints on the filter coefficients that generate linear phase as other filters of this class can be readily obtained from this form.

A. Constrained Weighted Least-Squares Technique

The conjugate-symmetric constraints are given by $h(n) = h^*(N - 1 - n)$, $n = 0, \cdots, N - 1$, where $\underline{h} = [h(0), h(1), \cdots, h(N-1)]^T$ represents the complex FIR filter coefficient vector.[1] We seek to obtain the coefficient vector $\underline{h}$ that minimizes the weighted integral squared-error criterion over the normalized frequency interval $[0, 1)$

$$J(\underline{h}) = \int_0^1 w(f)|z_D(f) - \underline{d}^H(f)\underline{h}|^2 df \qquad (1)$$

subject to the above conjugate-symmetric constraints. Here, $w(f)$ is a nonnegative frequency weighting function, $z_D(f)$ is the desired complex frequency response, $\underline{d}(f)$ is the frequency "steering" vector given by $\underline{d}(f) = [1, e^{j2\pi f}, \cdots, e^{j2\pi(N-1)f}]^T$, the inner product $\underline{d}^H(f)\underline{h}$ represents the filter frequency response, and f represents the actual frequency normalized by the sampling frequency. The objective function given by (1) can accommodate arbitrary desired multiband magnitude responses including zero weighted frequency intervals.

The conjugate-symmetric constraints can be compactly represented by $\underline{h}^* = E\underline{h}$, where E is the $N \times N$ exchange matrix with ones on the cross diagonal and zeros elsewhere. Note that $E = E^T$, and $E^2 = I$, where I is the identity matrix. Incorporation of this vector constraint via the complex Lagrange vector $\underline{\lambda}$ formulation yields the augmented objective function

$$J_1(\underline{h}) = J(\underline{h}) - \underline{\lambda}^T[\underline{h}^* - E\underline{h}] - \underline{\lambda}^H[\underline{h} - E\underline{h}^*]. \qquad (2)$$

Note that $J(\underline{h})$ and $J_1(\underline{h})$ are both real-valued functions for any complex vector $\underline{h}$. Expanding $J(\underline{h})$, differentiating with respect to $\underline{h}^*$ according to [10] (which treats a complex variable and its conjugate as independent variables) and equating to the null vector to satisfy the condition for the unique minimum yields

$$\frac{\partial J_1(\underline{h})}{\partial \underline{h}^*} = \int_0^1 \{ -w(f)z_D(f)\underline{d}(f) + w(f)\underline{d}(f)\underline{d}^H(f)\underline{h}\} df - \underline{\lambda} + E\underline{\lambda}^* = \underline{0}. \qquad (3)$$

Let

$$Q = \int_0^1 w(f)\underline{d}(f)\underline{d}^H(f) df \qquad (4)$$

$$\underline{u} = \int_0^1 w(f)z_D(f)\underline{d}(f) df. \qquad (5)$$

Note that Q is a Hermitian–Toeplitz matrix that is fully defined by either its first row or column. Use of (4) and (5) in (3) yields

$$Q\underline{h} - \underline{u} = \underline{\lambda} - E\underline{\lambda}^*. \qquad (6)$$

[1] The superscripts $*$, T, and H represent conjugate, transpose, and conjugate-transpose operations, respectively.
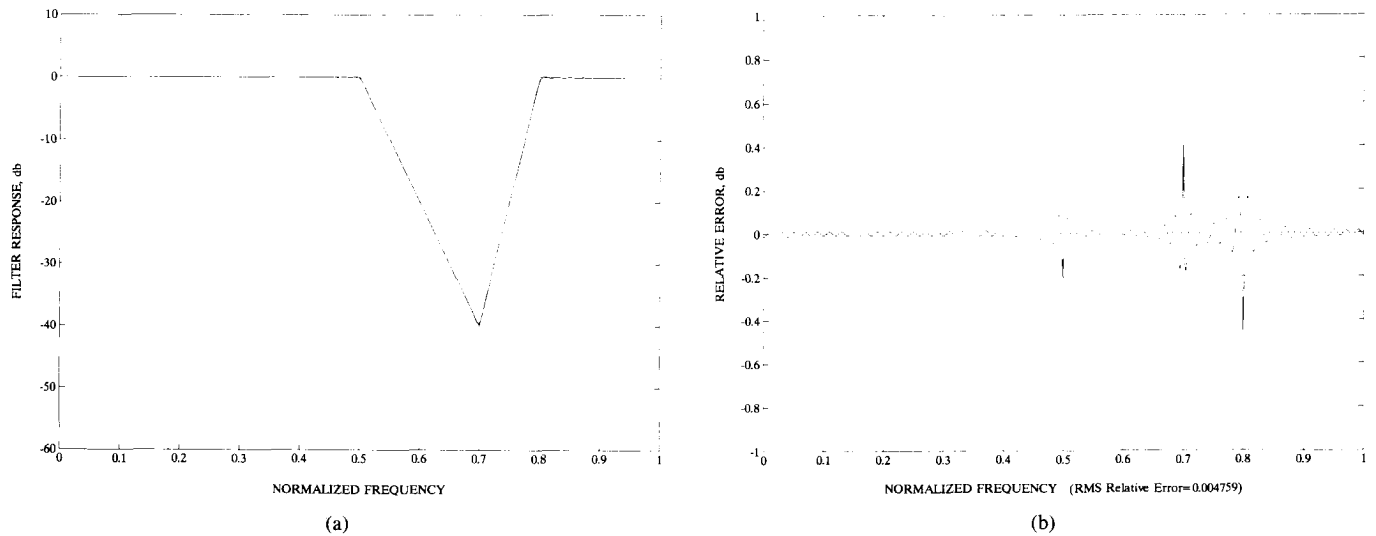
Fig. 1. Asymmetric v-notch filter design example: (a) Filter magnitude response; (b) relative error in magnitude response.

Let $\underline{\gamma} = \underline{\lambda} - E\underline{\lambda}^*$. Then

$$Q\underline{h} = \underline{u} + \underline{\gamma}. \tag{7}$$

We next determine $\underline{\gamma}$ so that the constraint $\underline{h}^* = E\underline{h}$ is satisfied. From (6) and (7), we have that

$$EQ\underline{h} = E\underline{u} + E\underline{\gamma} = E\underline{u} - \underline{\gamma}^*. \tag{8}$$

In addition, since Q is Hermitian-persymmetric, $EQ = Q^*E$ [11], and hence

$$Q^*E\underline{h} = E\underline{u} - \underline{\gamma}^*. \tag{9}$$

Subtracting (9) from the conjugate of (7) results in

$$Q^*[\underline{h}^* - E\underline{h}] = 2\underline{\gamma}^* + \underline{u}^* - E\underline{u}. \tag{10}$$

Applying the constraint $\underline{h}^* = E\underline{h}$ to (10) results in

$$\underline{\gamma} = \frac{1}{2}[E\underline{u}^* - \underline{u}]$$

and the solution for the filter coefficient vector as

$$Q\underline{h} = \frac{1}{2}[\underline{u} + E\underline{u}^*] \text{ or } \underline{h} = \frac{1}{2}Q^{-1}[\underline{u} + E\underline{u}^*]. \tag{11}$$

B. Unconstrained Weighted Least-Squares Technique

We derive here the unconstrained weighted least-squares complex FIR filter suitable for satisfying arbitrarily specified magnitude and phase responses (including nonlinear phase responses) that are necessary in many system applications. The solution immediately follows from the derivation in Section II-A by deleting the Lagrange multiplier constraints in (2), which results in $\underline{\gamma} = \underline{0}$ in (7), yielding the solution for the filter coefficient vector as

$$Q\underline{h} = \underline{u} \text{ or } \underline{h} = Q^{-1}\underline{u} \tag{12}$$

where Q and $\underline{u}$ are defined as before in (4) and (5).

C. Remarks

1) It can be readily verified that the constrained weighted least squares solution given by (11) does indeed satisfy the constraint $\underline{h}^* = E\underline{h}$, producing linear phase response, regardless of the desired complex response $z_D(f)$. The unconstrained solution given by (12) of course does not satisfy this property in general. However, it is of interest to note that if the desired response is

$z_D(f) = a_D(f)e^{j\phi_D(f)}$, where $a_D(f)$ is the desired magnitude response and $\phi_D(f)$ is linear phase with delay $\tau = (N - 1)/2$, then the two solutions become one and the same. This can be seen by substituting $z_D(f) = a_D(f)e^{-j2\pi f(N-1)/2}$ in the expression for $\underline{u}$ in (5) and simplifying, resulting in the nth element of $\underline{u}$ being given by

$$[\underline{u}]_n = \int_0^1 w(f)a_D(f)e^{j2\pi f(n-(N+1)/2)}\,df.$$

It can also be shown that the nth element of $E\underline{u}^*$ is given by the same expression, whereupon $E\underline{u}^* = \underline{u}$ and (11) becomes $\underline{h} = Q^{-1}\underline{u}$, which is the same as (12).

2) Since the matrix Q is Hermitian–Toeplitz and, hence, fully defined by either its first row or column, the solution for the filter coefficient vector can be obtained quickly and accurately in $o(N^2)$ operations by the Levinson recursion algorithm [11] as opposed to general matrix inversion methods, which require $o(N^3)$ operations. Furthermore, our methods obtain the *true* WLS solution, whereas the eigenfilter method [5], [7] obtains an approximate WLS solution that requires a variable number of iterations to compute the principal eigenvector (depending on the eigenvalue spread) and that necessitates $o(N^2)$ operations per iteration. Note also that as a special but useful case, the Q matrix in this correspondence reduces to a scalar multiple of the identity matrix when the weighting function is uniform and the desired amplitude response encompasses the entire frequency interval without unspecified frequency bands, allowing the solution of the coefficient vector to be obtained trivially.

3) Since the constrained technique results in a conjugate-symmetric filter, it would ostensibly appear computationally attractive to obtain the solution directly in terms of half the coefficient vector for even length filters. However, the resulting solution is actually more demanding computationally than the one presented here due to the more complicated and non-Toeplitz structure of the associated matrix of the system of linear equations (see also [12]).

III. ILLUSTRATIVE FILTER DESIGN EXAMPLES

In this section, we examine two filter design examples that illustrate the use of the constrained and unconstrained WLS techniques presented here. A linear-phase asymmetric v-notch filter design example suitable for radar/sonar clutter suppression applications is used to
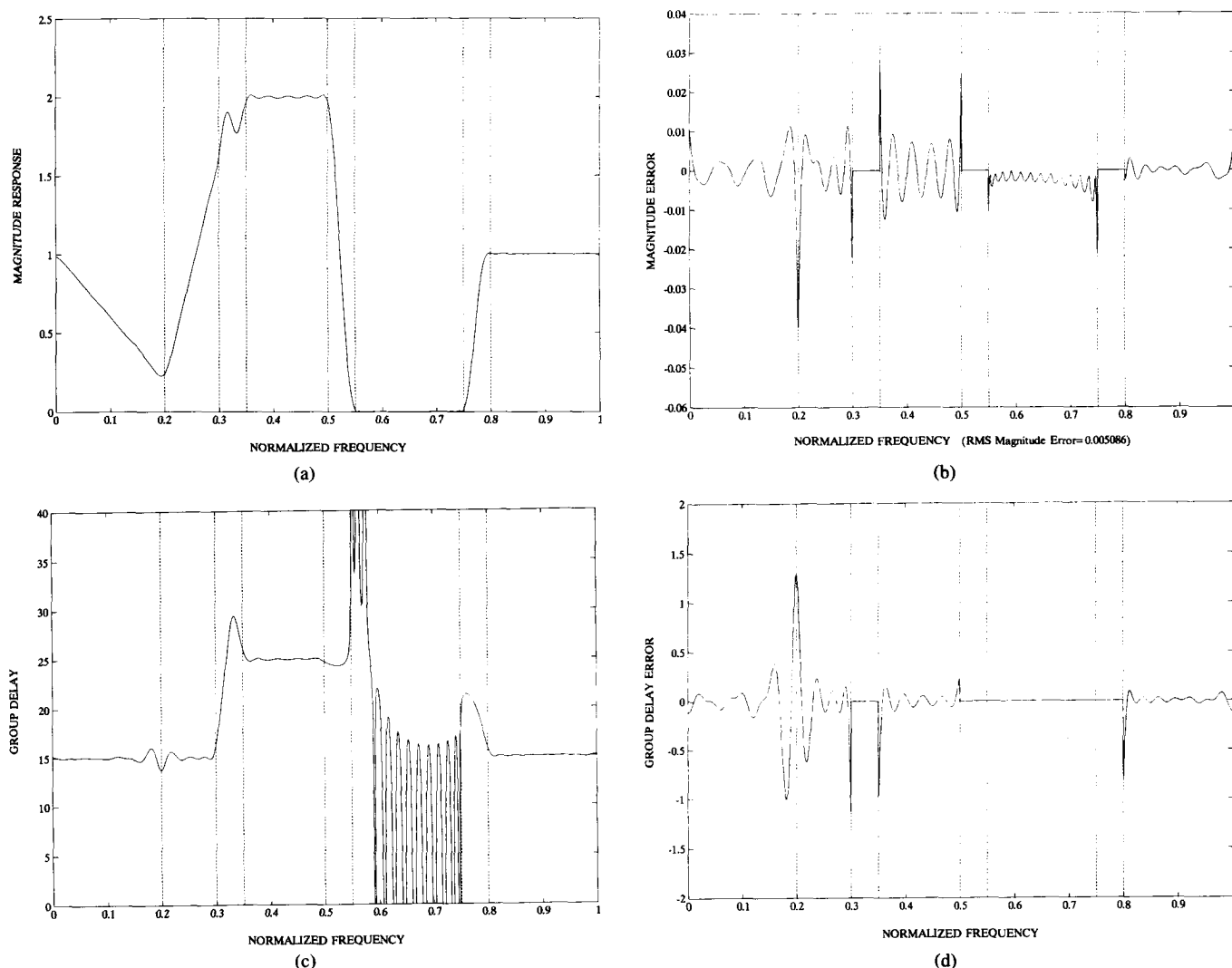
Fig. 2.   Arbitrary transfer function example of Nguyen [7] using the unconstrained WLS method of this correspondence: (a) Magnitude response; (b) error in magnitude response; (c) group delay; (d) error in group delay.

illustrate the use of the constrained technique, whereas a direct comparison with the results of Nguyen [7] for his arbitrary transfer function filter design example is used to illustrate the use of the unconstrained technique.

The techniques developed here necessitate the evaluation of certain integrals for the computation of Q and $\underline{u}$ given by (4) and (5). In general, these integrals would require numerical integration; however, for an important subclass of practical filter design problems (including all of the examples presented here), these integrals are readily evaluated in closed form. In particular, integrals arising from filter design problems specified by multisegment piecewise linear and exponential amplitude (linear in log-amplitude) response specifications with uniform or inverse squared-error weighting can be evaluated in closed form, resulting in improved numerical efficiency and accuracy.

The Linear Phase Asymmetric V-Notch Filter Design Problem:
The linear phase asymmetric v-notch filter design example is specified by the desired amplitude response function

$$|z_D(f)| = \begin{cases} 0 \text{ dB}, & 0 \leq f \prec 0.5 \\ [0, -40] \text{ dB}, & 0.5 \leq f \prec 0.7 \\ [-40, 0] \text{ dB}, & 0.7 \leq f \prec 0.8 \\ 0 \text{ dB}, & 0.8 \leq f \prec 1.0 \end{cases}$$

where the quantities in brackets specify the amplitudes at the endpoints of the exponential curve segment (linear in log-amplitude) that specifies the desired amplitude response in the frequency interval specified. The exponential amplitude response function is given by

$$|z_D(f)| = e^{\alpha_k + \beta_k f}, \text{ for } f \in [F_{k_1}, F_{k_2})$$

for the kth frequency interval $[F_{k_1}, F_{k_2})$ of the filter design specification. As the asymmetric v-notch filter has a 40-dB variation in its amplitude response, a minimum relative squared error estimation criterion is employed to balance the filter fit error amongst the specified frequency intervals evenly, resulting in the specification of the weighting function as

$$w(f) = \frac{1}{|z_D(f)|^2}.$$

A full derivation of the Q matrix and $\underline{u}$ vector for the relative squared error weighting and the linear and exponential amplitude response models is given in [9]. As this filter's amplitude response is asymmetric about any point in the normalized frequency domain (0 to 1 Hz.), it can only be generated with a complex FIR filter design technique; there is no purely real representation for these filter coefficients.

The amplitude response obtained by use of the constrained algorithm for the 101-tap complex linear phase FIR filter is given in Fig. 1(a), and the relative squared error, which is expressed in decibels, is given in Fig. 1(b). The constrained algorithm achieves a peak relative error of 0.41 dB at the frequency interval edges and a root-mean-square (RMS) error of 0.004759. The use of a relative squared-error minimization criterion is evident in the evenness of the error ripples across the large range of the filter's amplitude response Fig. 1(a).

The Arbitrary Filter Transfer Function Design Example of Nguyen [7]: Example 5, taken from Nguyen [7], is used to compare the unconstrained WLS technique presented here with the eigenfilter method of [7]. Nguyen's example consists of a specification with four passbands and one stopband with specified amplitude and phase requirements that, due to its asymmetry, necessitates a complex FIR filter synthesis technique. The unspecified frequency intervals are unweighted and do not contribute to the total fit error. Nguyen's example is specified with an absolute squared-error optimization criterion rather than the relative squared-error criterion used previously. The amplitude response attained by the unconstrained WLS technique for a 50-tap FIR filter is given in Fig. 2(a), the amplitude error in Fig. 2(b), the group delay in Fig. 2(c), and the group delay error in Fig. 2(d). The corresponding RMS errors are also shown in the figures. While the results obtained here are nearly identical to those of Nguyen, they represent the true WLS solution, which is also computed much more efficiently than the eigenfilter technique of [7] (see also Remark 2).

REFERENCES

[1] T. W. Parks and J. H. McClellan, "Chebyshev approximations for nonrecursive digital filters with linear phase," *IEEE Trans. Circuit Theory*, vol. CT-19, no. 2, pp. 189–194, Mar. 1972.

[2] L. R. Rabiner and B. Gold, *Theory and Applications of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1975.

[3] X. Chen and T. W. Parks, "Design of FIR filters in the complex domain," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-35, no. 2, pp. 144–153, Feb. 1987.

[4] Y. C. Lim, J. H. Lee, C. K. Chen, and R. H. Yang, "A weighted least squares algorithm for quasiequiripple FIR and IIR digital filter design," *IEEE Trans. Signal Processing*, vol. 40, no. 3, pp. 551–558, Mar. 1992.

[5] P. P. Vaidyanathan and T. Q. Nguyen, "Eigenfilters: A new approach to least-squares FIR filter design and applications including Nyquist filters," *IEEE Trans. Circuits Syst.*, vol. CAS-34, pp. 11–23, Jan. 1987.

[6] K. Preuss, "On the design of FIR filters by complex Chebyshev approximation," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 37, no. 5, pp. 702–712, May 1989.

[7] T. Q. Nguyen, "The design of arbitrary FIR digital filters using the eigenfilter method," *IEEE Trans. Signal Processing*, vol. 41, no. 3, pp. 1128–1139, Mar. 1993.

[8] S. C. Pei and J. J. Shyu, "Complex eigenfilter design of arbitrary complex coefficient FIR digital filters," *IEEE Trans. Circuits Syst.*, vol. 40, no. 1, pp. 32–40, Jan. 1993.

[9] A. G. Jaffer and W. E. Jones, "Constrained least-squares design and characterization of affine phase complex FIR filters," in *Proc. 27th Ann. Asilomar Conf. Signals, Syst., Comput.*, Nov. 1993, pp. 685–691.

[10] D. H. Brandwood, "A complex gradient operator and its applications in adaptive array theory," *Proc. Inst. Elec. Eng.*, vol. 130, pts. F and H, no. 1, pp. 11–16, Feb. 1983.

[11] S. L. Marple, *Digital Spectral Analysis with Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

[12] A. G. Jaffer, W. E. Jones, and T. J. Abatzoglou, "Weighted least-squares design of linear-phase and arbitrary 2-D complex FIR filters," presented at the *1995 IEEE Int. Conf. Acoust., Speech, Signal Processing* (ICASSP-95), May 1995.

Constraints on the Cutoff Frequencies of Mth-Band Linear-Phase FIR Filters

James M. Nohrden and Truong Q. Nguyen

Abstract—In this correspondence, constraints are derived for the cutoff frequencies of linear-phase FIR Mth-band filters such that the filters have good passband and stopband characteristics, i.e. ones that very closely approximate an ordinary (non Mth-band) filter designed using some optimal method. Constraints on lowpass filters are first considered, and the results are extended to multiband filters.

I. INTRODUCTION

Mth-band filters have found numerous applications in recent years [2]–[4], [9], [11], [14], [15]. In signal processing, Mth-band filters are used in 1-D [15] and 2-D [2] perfect reconstruction filter banks, nonuniform sampling [4], interpolation filters [14], and intersymbol interference rejection [11]. Additionally, Mth-band filters have found applications in antenna array design [3]. Mth-band filters are commonly designed as lowpass filters with cut-off frequencies at π/M. This does not have to be the case. In fact, bandpass and multiband Mth-band filters may be designed using the constrained set of cut-off frequencies derived in this paper.

Fig. 1 shows the desired response of a lowpass filter where ω_p and ω_s are the passband and stopband cutoff frequencies, respectively. δ_p and δ_s are the corresponding errors. The center frequency ω_c of a lowpass filter is defined as

$$\omega_c = \frac{\omega_p + \omega_s}{2}. \tag{1}$$

Let $\hat{H}(z)$ denote the transfer function of an odd length linear-phase FIR filter

$$\hat{H}(z) = \sum_{n=0}^{N-1} \hat{h}(n)z^{-n}, \quad \hat{h}(n) \text{ real} \tag{2}$$

and define a noncausal shifted version of $\hat{H}(z)$ as $H(z) = z^L \hat{H}(z)$, where $L = (N-1)/2$. $H(z)$ is more suitable for the analytical work in this correspondence, whereas $\hat{H}(z)$ is actually implemented.

Optimal design techniques exist to minimize the frequency domain error for linear-phase FIR filters. One such example is the Remez algorithm [5], which minimizes the maximum error and therefore has an equiripple frequency response. Another algorithm is the eigenfilter approach [13], which minimizes the least squares error.

Let us define a good Mth-band filter as one that has approximately the same passband and stopband error characteristics as a non-Mth-band optimal filter with the same specifications. In other words, a good Mth-band filter is an Mth-band filter that is very nearly an optimal filter.

J. M. Nohrden is with Massachusetts Institute of Technology, Lincoln Laboratory, Lexington, MA 02173-9108.

T. Q. Nguyen is with the Department of Electrical and Computer Engineering, University of Wisconsin, Madison, Madison, WI 53706 USA.

```
/////////////////////////////////////////////////////////////////////////////////////////
//  FilePath: CLSLP_20240920_191907/OTHER_CLSLP_EXAMPLES/Filter_1_eExp_RelErr_mid_seg/Start.cpp
//  Tag:       CLSLP_20240921_1739
//  TimeDate:  20240921_183231
//  Email:     WEJC@WEJC.COM
//  Author:    William Earl Jones
/////////////////////////////////////////////////////////////////////////////////////////
//                         Copyright (c) 2024 William Earl Jones                        //
//                                                                                      //
//                          ---Standard BSD 3-Clause License---                         //
//                                                                                      //
//  Redistribution and use in source and binary forms, with or without modification, are //
//  permitted provided that the following conditions are met:                           //
//                                                                                      //
//     1. Redistributions of source code must retain the above copyright notice, this list of //
//        conditions and the following disclaimer.                                      //
//                                                                                      //
//     2. Redistributions in binary form must reproduce the above copyright notice, this list of //
//        conditions and the following disclaimer in the documentation and/or other materials //
//        provided with the distribution.                                               //
//                                                                                      //
//     3. Neither the name of the copyright holder nor the names of its contributors may be used //
//        to endorse or promote products derived from this software without specific prior //
//        written permission.                                                           //
//                                                                                      //
//  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS //
//  OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF      //
//  MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE //
//  COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, //
//  EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF   //
//  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) //
//  HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR //
//  TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, //
//  EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.                                   //
/////////////////////////////////////////////////////////////////////////////////////////<wejc>////


#include  <fftw3.h>
#include  <iostream>
#include  <fstream>
#include  <chrono>
#include  "CLSLP.h"


double  BlackmanHarrisWindow( const double K, const double N );
```

```cpp
int  main( int /* NumArgs */, char** /* ArgList */ )
{
  const uint32_t          FftSize     = 1U << 12;
  const uint32_t          NumFiltCoefs = 501U;
  vector<complex<double>>  OutCoefs(NumFiltCoefs);

  std::ofstream  OutFiltCoefs("FilterCoefs.dat");
  std::ofstream  OutSpectral ("Spectral.dat");
  std::ofstream  OutPhase    ("Phase.dat");

  std::cout << ">>[FilterSynthesis: Example Filters]" << std::endl;

  {
    // Generate the filter
    CLSLP  Filt;

    // â\200\230AddSymmetricâ\200\231 is for Real-Valued Filter Coefficients    ([0,.5] mod 1.) Normalized Hz
    // â\200\230Addâ\200\231         is for Complex-Valued Filter Coefficients ([0,1.) mod 1.) Normalized Hz
    Filt.AddSymmetric( CLSLP::eLin_AbsErr, .0, .1, 0.00, 0.00, 1 );
    Filt.AddSymmetric( CLSLP::eExp_RelErr, .1, .4, 1e-4, 1.00, 1 );
    Filt.AddSymmetric( CLSLP::eLin_AbsErr, .4, .5, 0.00, 0.00, 1 );

    // Generate the filter coefficients
    const auto Clock_0 = chrono::high_resolution_clock::now();

    // Calculate the FIR Filter Coeffcients
    Filt.GenFilter( NumFiltCoefs, OutCoefs );

    const auto Clock_1 = chrono::high_resolution_clock::now();

    for( auto K=0U ; K < NumFiltCoefs ; ++K )
    {
      OutFiltCoefs << OutCoefs[K] << std::endl;
    }

    const double DeltaTime =
        double(chrono::duration_cast <chrono::microseconds> (Clock_1 - Clock_0).count());

    cout << "--FIR Filter Coefficient Generation: NumCoefs:"
         << NumFiltCoefs << "  [TotalSynthPeriod:"
         << DeltaTime << " uS  PerCoefSynthPeriod:"
         << (DeltaTime/NumFiltCoefs) << " uS/Coef]"
```

```cpp
                  << std::endl;

        cout << "--Plot Generation" << std::endl;

        // Take DFT of these coefficents
        {
          fftw_complex  *In=nullptr, *Out=nullptr;
          fftw_plan      Plan=nullptr;

          In  = (fftw_complex*) fftw_malloc( sizeof(fftw_complex) * FftSize );
          Out = (fftw_complex*) fftw_malloc( sizeof(fftw_complex) * FftSize );

          // Plan the FFT
          Plan = fftw_plan_dft_1d( FftSize, In, Out, FFTW_FORWARD, FFTW_PATIENT );

          // Fill input data.
          //   No need to iterate on [0,FftSize).  Only (0,NumFiltCoefs) non-zero.
          for( auto K=0U ; K < NumFiltCoefs ; ++K )
          {
            const double  BHW = BlackmanHarrisWindow( K, NumFiltCoefs );

            In[K][0] = BHW * real(OutCoefs[K]);
            In[K][1] = BHW * imag(OutCoefs[K]);
          }

          for( auto K=NumFiltCoefs ; K < FftSize ; ++K )
          {
            In[K][0] = 0;
            In[K][1] = 0;
          }

          // Perform the FFT
          fftw_execute(Plan);

          for( int32_t  K = -(FftSize>>1) ; K < (signed) (FftSize>>1) ; ++K )
          {
            const double  A = ( (double) K / FftSize );
            double        Mag = 0;

            // Amplitude
            if( K >= 0 )
            {
              Mag = 10.*log10(Out[K][0]*Out[K][0] + Out[K][1]*Out[K][1]);
            }
```

```cpp
      else
      {
        assert( ( K + (signed) FftSize ) >= 0 );

        const uint32_t  K2 = K + FftSize;

        Mag = 10.*log10(Out[K2][0]*Out[K2][0] + Out[K2][1]*Out[K2][1]);
      }

      OutSpectral <<  A  << "  " << Mag << std::endl;

      // Phase Response
      double  Phase = 0;

      // Phase
      if( K >= 0 )
      {
        Phase = atan2( Out[K][1], Out[K][0] );
      }
      else
      {
        Phase = atan2( Out[K+FftSize-1U][1], Out[K+FftSize-1U][0] );
      }

      OutPhase <<  A  << "  " << Phase << std::endl;
    }

    // Clean Up
    fftw_destroy_plan(Plan);
    fftw_free(In);
    fftw_free(Out);
  }
}

  std::cout << "<<[FilterSynthesis]" << std::endl;

  return  0;
}


double  BlackmanHarrisWindow( const double Num, const double Den )
{
  const double  Mult = M_PI * Num / Den;
```

```
  const double  BHD =   0.35875
                      − 0.48829 * cos(2*Mult)
                      + 0.14128 * cos(4*Mult)
                      − 0.01168 * cos(6*Mult);

  return  BHD;
}
```

```
////////////////////////////////////////////////////////////////////////////////////////
//  FilePath:  CLSLP_20240920_191907/CLSLP/CLSLP.h
//  Tag:       CLSLP_20240921_1739
//  TimeDate:  20240921_183231
//  Email:     WEJC@WEJC.COM
//  Author:    William Earl Jones
////////////////////////////////////////////////////////////////////////////////////////
//                        Copyright (c) 2024 William Earl Jones                        //
//                                                                                     //
//                        ---Standard BSD 3-Clause License---                          //
//                                                                                     //
//  Redistribution and use in source and binary forms, with or without modification, are //
//  permitted provided that the following conditions are met:                          //
//                                                                                     //
//     1. Redistributions of source code must retain the above copyright notice, this list of //
//        conditions and the following disclaimer.                                     //
//                                                                                     //
//     2. Redistributions in binary form must reproduce the above copyright notice, this list of //
//        conditions and the following disclaimer in the documentation and/or other materials //
//        provided with the distribution.                                              //
//                                                                                     //
//     3. Neither the name of the copyright holder nor the names of its contributors may be used //
//        to endorse or promote products derived from this software without specific prior //
//        written permission.                                                          //
//                                                                                     //
//  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS //
//  OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF     //
//  MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE //
//  COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, //
//  EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF  //
//  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) //
//  HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR //
//  TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, //
//  EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.                                  //
/////////////////////////////////////////////////////////////////////////////////<wejc>////

#pragma once

#include  <complex>
#include  <vector>
#include  <sstream>
#include  <list>
#include  <iomanip>
#include  <cassert>
```

```
#include  <cstdint>
#include  <gsl/gsl_sf.h>

using namespace std;


/*//////////////////////////////////////////////////////////////////////////////////////////////
  class  CLSLP

  The CLSLP C++ class is a tool that generates the coefficients for a linear-phase FIR
  filter filters. These filters are specified in line segments of frequency and amplitude
  (i.e. freq:[F0,F1], amp: [A0,A1]).  Two types of segments: linear and exponential.  Linear
  segmentâ\200\231s amplitude are specified by an equation of the form B*f+A where B and A are picked
  so that the line-segment end points, freq:[F0,F1] and amp: [A0,A1], are satisfied.  The other
  type of segment is the exponential segment defined by exp(B*f+A).  In this case the end points
  are specified by the user as freq:[F0,F1], amp: [loge(A0),loge(A1)].  Please read the PDF that
  accompanies this software for examples of use and mathematical background.  These algorithms
  were developed and published by Amin G. Jaffer and William E. Jones in the 1994 timeframe.

  The Diagnostics member function gives an example of each segment type and the code associated
  with FIR filter coefficient generation and evaluation.
//////////////////////////////////////////////////////////////////////////////////////////////*/

class  CLSLP
{
public:

  // Some constants
  static const double  TPI;
  static const double  PI2;

  // Segment types
  //  The interval is defined by [F0,F1] in frequency and [A0,A1] in amplitude for Linear Segments
  //  The interval is defined by [F0,F1] in frequency and [loge(A0),loge(A1)] in amplitude for
  //       Exponential Segments.

  //  Note that frequencies are specified in the interval [0,1).  1/2 is PRF/2 at all sample rates.
  typedef enum  eSegType
  {
    eExp_RelErr,    // exp(B*f+A) segment, relative error
    eExp_AbsErr,    // exp(B*f+A) segment, absolute error
    eLin_RelErr,    // B*f+A segment, relative error
    eLin_AbsErr     // B*f+A segment, absolute error
  } SegType;
```

```cpp
public:

  // Exponential to a imaginary power
  static inline complex<double> ExpI( const double ImagPart )
  {
    return  complex<double>( cos(ImagPart), sin(ImagPart) );
  }

  // The E1 exponential integral
  static inline complex<double> ExpInt_1_I( const double ImagArg )
  {
    return  complex<double>( -gsl_sf_Ci(fabs(ImagArg)), -.5*M_PI+gsl_sf_Si(ImagArg) );
  }

  // The E2 exponential integral
  static inline complex<double> ExpInt_2_I( const double Arg )
  {
    return  ( ExpI(-Arg) - MultI( Arg * ExpInt_1_I( Arg ) ) );
  }

  // Multiply by i
  static inline complex<double> MultI( const complex<double> Arg )
  {
    return  complex<double>( -imag(Arg), real(Arg) );
  }

  //>> START class Seg
  class  Seg
  {
  public:

    // Linear Segments are: B*f+A.  The interval is defined by [F0,F1] in frequency and [A0,A1] in amplitude.
    // Exponential Segments are: exp(B*f+A).  The interval is defined by [F0,F1] in frequency and
    //                                      [loge(A0),loge(A1)] in amplitude.
    //  Note that frequencies are specified in the interval [0,1).  1/2 is PRF/2 at all sample rates.
    SegType  Type;
    double   F0, F1, A0, A1, W;

    Seg( const SegType Type_Param,
         const double  F0_Param, const double  F1_Param,
         const double  A0_Param, const double  A1_Param,
         const double  Weight_Param=1. )
    {
```

```
      Type = Type_Param;
      F0   = F0_Param;
      F1   = F1_Param;
      A0   = A0_Param;
      A1   = A1_Param;
      W    = Weight_Param;
    }

    // This call modifies the Q matrix and R vectors to account for this segment.  It in turn calls
    //    specific generator functions based on the type of segment desired.
    //   Linear Segments are: B*f+A.  The interval is defined by [F0,F1] in frequency and [A0,A1] in amplitude.
    //    Exponential Segments are: exp(B*f+A).  The interval is defined by [F0,F1] in frequency and
    //                                            [loge(A0),loge(A1)] in amplitude.
    //  Note that frequencies are specified in the interval [0,1].  1/2 is PRF/2 at all sample rates.
    void   Gen_Segment( vector<complex<double>>&  Q,
                        vector<complex<double>>&  R,
                        const double              Weight=1. );


    // Display a segment definition
    string   Display( const double SampleRate=1 ) const
    {
      ostringstream  oS;

      oS << "SEG[";

      switch(Type)
      {
      case eExp_RelErr:
        oS << "Exp_Rel";
      break;

      case eExp_AbsErr:
        oS << "Exp_Abs";
      break;

      case eLin_RelErr:
        oS << "Lin_Rel";
      break;

      case eLin_AbsErr:
        oS << "Abs_Lin";
      break;

      default:
```

```
        oS << "Error";
        assert(false);
      break;
      }

    oS  << "] "
        << setiosflags(ios::fixed) << setprecision(6) << setw(12) << (this->F0*SampleRate)
        << ": "
        << setiosflags(ios::fixed) << setprecision(8) << setw(12) << this->A0
        << "  <--->  "
        << setiosflags(ios::fixed) << setprecision(6) << setw(12) << (this->F1*SampleRate)
        << ": "
        << setiosflags(ios::fixed) << setprecision(8) << setw(12) << this->A1
        << "  W:"
        << setiosflags(ios::fixed) << setprecision(8) << setw(12) << this->W
        << "]";

    return  oS.str();
    }

protected:

    // exp(B*f+A) segment, relative error
    // -Note that frequencies are specified in the interval [0,1).  1/2 is PRF/2 at all sample rates.
    void  Gen_ExpRel_Segment( vector<std::complex<double>>&  Q,
                              vector<std::complex<double>>&  R,
                              const double                   Weight );

    // exp(B*f+A) segment, relative error
    // -Note that frequencies are specified in the interval [0,1).  1/2 is PRF/2 at all sample rates.
    void  Gen_ExpAbs_Segment( vector<std::complex<double>>&  Q,
                              vector<std::complex<double>>&  R,
                              const double                   Weight );

    // B*f+A segment, relative error
    // -Note that frequencies are specified in the interval [0,1).  1/2 is PRF/2 at all sample rates.
    void  Gen_LinRel_Segment( vector<std::complex<double>>&  Q,
                              vector<std::complex<double>>&  R,
                              const double                   Weight );

    // B*f+A segment, absolute error
    // -Note that frequencies are specified in the interval [0,1).  1/2 is PRF/2 at all sample rates.
    void  Gen_LinAbs_Segment( vector<std::complex<double>>&  Q,
                              vector<std::complex<double>>&  R,
```

```
                                    const double                    Weight );
  };
  //<< END class  Seg

  // Linear system solution of 'matrix(Q) vector(Coefs) = vector(R)' via generalized Levenson's recusion
  bool  Levenson( vector<complex<double>>&  pR,
                  vector<complex<double>>&  g,
                  vector<complex<double>>&  X  );

  complex<double>  InnerProdR( const uint32_t                 Start,
                              const uint32_t                 StopPlus1,
                              const vector<complex<double>>&  InData1,
                              const vector<complex<double>>&  InData2    ) const;

  complex<double>  InnerProdC( const uint32_t                 Start,
                              const uint32_t                 StopPlus1,
                              const vector<complex<double>>&  InData1,
                              const vector<complex<double>>&  InData2    ) const;

private:

  // The segments list
  list<Seg>  Segs;

  double  Gain;   // Default amplitude (1) unless set by '.SetAmplitude(xx)'

public:

  CLSLP(void);

  void SetGain( const double GainParam )
  {
    this->Gain = GainParam;
  }

  double GetGain(void) const
  {
    return  this->Gain;
  }

  // Add a new filter segment
  //    Linear Segments are: B*f+A.  The interval is defined by [F0,F1] in frequency and [A0,A1] in amplitude.
  //    Exponential Segments are: exp(B*f+A).  The interval is defined by [F0,F1] in frequency and
  //                                           [loge(A0),loge(A1)] in amplitude.
```

```cpp
  //  Note that frequencies are specified in the interval [0,1).  1/2 is SampleRate/2 here.
  void  Add( const SegType Type,
             const double  F0, const double  F1,
             const double  A0, const double  A1,
             const double  Weight=1. )
  {
    assert( F0 < F1 );

    // Add: Design for a assymetric filter.  This type of filter will process the real and imaginary channels
    //      with different coefficients.  [To generate a Symmetric, real-valued, filter, specifiy desired
    //      filter responses in a conjugate-symmetric pattern about SampleRate/2.]
    // ===  by the design.
     Segs.push_back( Seg( Type, F0, F1, A0*this->Gain, A1*this->Gain, Weight ) );
  }

  // Add a new filter segment symmetric between the positive andnegative frequencies
  //   Linear Segments are: B*f+A.  The interval is defined by [F0,F1] in frequency and [A0,A1] in amplitude.
  //   Exponential Segments are: exp(B*f+A).  The interval is defined by [F0,F1] in frequency and
  //                                          [loge(A0),loge(A1)] in amplitude.
  //  Note that frequencies are specified in the interval [0,1).  1/2 is SampleRate/2 at all sample rates.
  void  AddSymmetric( const SegType Type,
                      const double  F0, const double  F1,
                      const double  A0, const double  A1,
                      const double  Weight=1. )
  {
    assert( F0 <= .5 );
    assert( F0 <= .5 );
    assert( F0 <  F1 );

    // AddSymmetric: Filter response is conjugate-symmetric about PRF/2 and linear-phase response is assumed
    // ============  by the fiter design algorithm.
    Segs.push_back( Seg( Type,   F0,   F1, A0*this->Gain, A1*this->Gain, Weight ) );
    Segs.push_back( Seg( Type, 1-F1, 1-F0, A1*this->Gain, A0*this->Gain, Weight ) );
  }

  // Generate the filter coefficients out into Out.  This involves the solution of the linear system
  //    'matrix(Q) vector(Coefs) = vector(R)'.
  bool  GenFilter( const unsigned int FilterLen, vector<complex<double>>& Out );

  // Get the number of filter segments
  unsigned int  GetNumSegments(void) const
  {
    return  Segs.size();
  }
```

```
  // Clear all filter segments
  void  Clear(void)
  {
    Segs.clear();
  }

  // Get segment list reference
  list<Seg>&  GetSegs(void)
  {
    return  this->Segs;
  }
};
```

```
///////////////////////////////////////////////////////////////////////////////////////
//  FilePath:  CLSLP_20240920_191907/CLSLP/CLSLP.cpp
//  Tag:       CLSLP_20240921_1739
//  TimeDate:  20240921_183231
//  Email:     WEJC@WEJC.COM
//  Author:    William Earl Jones
///////////////////////////////////////////////////////////////////////////////////////
```

```
#include  <cassert>
#include  <cmath>
#include  <fstream>
#include  <iostream>
#include  "CLSLP.h"

using namespace std;
```

```cpp
const double  CLSLP::TPI = 2. * M_PI;
const double  CLSLP::PI2 = M_PI * M_PI;



CLSLP::CLSLP(void) : Gain(1)
{
}


// Generate the filter coefficients into Out
bool  CLSLP::GenFilter( const unsigned int        FilterLen,
                        vector<complex<double>>&  Out  )
{
  // Is zero length
  if( FilterLen == 0 )  return false;

  vector<complex<double>>  Q(FilterLen);
  vector<complex<double>>  R(FilterLen);

  // Over The Frequency Segments
  auto  I = Segs.begin();

  for(  ; I != Segs.end() ; ++I )
  {
    I->Gen_Segment( Q, R );
  }

  // Perform Levenson To Solve The System
  const bool  Solution = Levenson( Q, R, Out );

  return  Solution;
}


// Fill the Q-matrix and the R-vector for linear system solution to filter coefficients
void  CLSLP::Seg::Gen_Segment( vector<complex<double>>&  Q,
                               vector<complex<double>>&  R,
                               const double              Weight )
{
  // Pick the frequency interval segment (line) type
  switch(this->Type)
  {
    case eExp_RelErr:
```

```cpp
      this->Gen_ExpRel_Segment( Q, R, Weight );
    break;

    case eExp_AbsErr:
      this->Gen_ExpAbs_Segment( Q, R, Weight );
    break;

    case eLin_RelErr:
      this->Gen_LinRel_Segment( Q, R, Weight );
    break;

    case eLin_AbsErr:
      this->Gen_LinAbs_Segment( Q, R, Weight );
    break;

    default:
      assert(false);   // Set exception in DEBUG mode
    break;
  }
}


// exp(B*f+A) segment, relative error
void  CLSLP::Seg::Gen_ExpRel_Segment( vector<complex<double>>&  Q,
                                      vector<complex<double>>&  R,
                                      const double              Weight  )
{
  const double     Min_Segment = 1e-32;   // Need > 0 here
  const double     MidPoint    = .5 * ( Q.size() - 1 );
  double           lA1         = 0;
  double           lA0         = 0;
  complex<double>  Ret { 0 };

  if( this->A1 >= Min_Segment )
  {
    lA1 = log(this->A1);
  }
  else
  {
    lA1 = log(Min_Segment);
  }

  if( this->A0 >= Min_Segment )
  {
```

```cpp
    lA0 = log(this->A0);
  }
  else
  {
    lA0 = log(Min_Segment);
  }

  // Put in standard form
  const double  sB  = (lA1  - lA0) / (this->F1 - this->F0);
  const double  sA  = lA0 - sB  * this->F0;

  // Form The Q Toeplitz Matrix (Vector Here)
  for( auto L=0U ; L < Q.size() ; ++L )
  {
    if( !((L==0) && (sB == 0)) )
    {
      Ret   = ExpI(TPI*L*this->F1) * exp(-2.*(sB*this->F1+sA));
      Ret  -= ExpI(TPI*L*this->F0) * exp(-2.*(sB*this->F0+sA));

      Q[L] +=  Weight * Ret / complex<double>( -2.*sB, TPI*L );
    }
    else
    {
      Q[L] +=  Weight * exp(-2.*sA) * (this->F1 - this->F0);
    }
  }

  // Form The R Vector
  for( auto L=0U ; L < R.size() ; ++L )
  {
    if( !((L==MidPoint) && (sB == 0)) )
    {
      Ret  = ExpI(-TPI*this->F1*(L-MidPoint)) * exp(-(sB*this->F1+sA));
      Ret -= ExpI(-TPI*this->F0*(L-MidPoint)) * exp(-(sB*this->F0+sA));

      R[R.size()-1-L] +=  Weight * Ret / complex<double>( -sB, -TPI*(L-MidPoint));
    }
    else
    {
      R[R.size()-1-L] +=  Weight * exp(sA) * (this->F1 - this->F0);
    }
  }
}
```

```cpp
// exp(B*f+A) segment, absolute error
void  CLSLP::Seg::Gen_ExpAbs_Segment( vector<complex<double>>&  Q,
                                      vector<complex<double>>&  R,
                                      const double              Weight  )
{
  const double      Min_Segment = 1e-32;    // Need > 0 here
  const double      MidPoint    = .5 * ( Q.size() - 1 );
  double            lA1         = 0;
  double            lA0         = 0;
  complex<double>  Ret { 0 };

  if( this->A1 >= Min_Segment )
  {
    lA1 = log(this->A1);
  }
  else
  {
    lA1 = log(Min_Segment);
  }

  if( this->A0 >= Min_Segment )
  {
    lA0 = log(this->A0);
  }
  else
  {
    lA0 = log(Min_Segment);
  }

  // Put in standard form
  const double  sB  = (lA1  - lA0) / (this->F1 - this->F0);
  const double  sA  = lA0 - sB  * this->F0;

  // Form The Q Toeplitz Matrix (Vector Here)
  for( auto L=0U ; L < Q.size() ; ++L )
  {
    // See if we can assume a zero denominator
    if( L != 0 )
    { // L != 0
      Ret  = -ExpI( TPI*L*this->F1 );
      Ret -= -ExpI( TPI*L*this->F0 );

      Q[L]  +=  Weight * MultI( Ret ) / (TPI*L);
```

```cpp
    }
    else
    { // L == 0
      Q[L]  +=  Weight * ( this->F1 - this->F0 );
    }
  }


  // Form The R Vector
  for( auto L=0U ; L < R.size() ; ++L )
  {
    if( !((L == MidPoint) && (fabs(sB) <= Min_Segment)) )
    { // Not midpoint
      Ret  =  ExpI(-TPI*this->F1*(L-MidPoint)) * exp(sB*this->F1+sA);
      Ret -=  ExpI(-TPI*this->F0*(L-MidPoint)) * exp(sB*this->F0+sA);

      R[R.size()-1-L] += Weight * Ret / complex<double>( sB, -TPI*(L-MidPoint) );
    }
    else
    { // Midpoint
      R[R.size()-1-L] += Weight * ( this->F1 - this->F0 ) * exp(sA);
    }
  }
}


// B*f+A segment, relative error
void  CLSLP::Seg::Gen_LinRel_Segment( vector<complex<double>>&  Q,
                                      vector<complex<double>>&  R,
                                      const double              Weight  )
{
  const double    MidPoint = .5 * ( Q.size() - 1 );
  double          Phase0 = 0;
  double          Phase1 = 0;
  complex<double> Ret { 0 };

  // Put in standard form
  const double  sB = (this->A1  - this->A0) / (this->F1 - this->F0);
  const double  sA = this->A0 - sB  * this->F0;

  // Form The Q Toeplitz Matrix (Vector Here)
  for( auto L=0U ; L < Q.size() ; ++L )
  {
    // Handle special cases
    if( L != 0 )
```

```
  { // L != 0
    if( sB != 0 )
    { // B != 0  &&  L != 0
      Phase1 = -TPI*L*(sB*this->F1+sA) / sB;
      Phase0 = -TPI*L*(sB*this->F0+sA) / sB;

      Ret   =  -ExpInt_2_I(Phase1) / (sB*(sB*this->F1+sA));
      Ret  -=  -ExpInt_2_I(Phase0) / (sB*(sB*this->F0+sA));

      Q[L]  +=  Weight * Ret * ExpI(-TPI*sA*L/sB);
    }
    else
    { // B == 0  &&  L != 0
      Ret   =  -MultI( ExpI(TPI*L*this->F1) );
      Ret  -=  -MultI( ExpI(TPI*L*this->F0) );

      Q[L] +=  Weight * Ret / (TPI*sA*sA*L);
    }
  }
  else
  { // L == 0  &&  B != 0
    if( sB != 0 )
    { // B!= 0
      Ret   =  -1. / (sB*(sB*this->F1+sA));
      Ret  -=  -1. / (sB*(sB*this->F0+sA));

      Q[L] +=  Weight * Ret;
    }
    else
    { // L == 0  &&  B == 0
      Q[L] +=  Weight * (this->F1 - this->F0) / (sA*sA);
    }
  }
}

// Form The R Vector (Vector Here)
for( auto L=0U ; L < R.size() ; ++L )
{
  // Handle special cases
  if( L != MidPoint )
  { // L != MidPoint
    if( sB != 0 )
    { // B != 0  &&  L != MidPoint
      Phase1 = TPI * (sB*this->F1+sA)*(L-MidPoint) / sB;
```

```cpp
          Phase0 = TPI * (sB*this->F0+sA)*(L-MidPoint) / sB;

          Ret    =  -ExpInt_1_I(Phase1);
          Ret    -=  -ExpInt_1_I(Phase0);

          R[R.size()-1-L]  +=  Weight * Ret * ExpI(TPI*sA*(L-MidPoint)/sB) / sB;
        }
        else
        { // B == 0  &&  L != MidPoint
          Ret    =  MultI( ExpI(-TPI*this->F1*(L-MidPoint)));
          Ret    -=  MultI( ExpI(-TPI*this->F0*(L-MidPoint)));

          R[R.size()-1-L]  +=  Weight * Ret / (TPI*sA*(L-MidPoint));
        }
      }
      else
      { // L == MidPoint  &&  B != 0
        if( sB != 0 )
        { // B != 0
          Ret  =  log(sB*this->F1+sA);
          Ret -=  log(sB*this->F0+sA);

          R[R.size()-1-L]  +=  Weight * Ret / sB;
        }
        else
        { // L == MidPoint  &&  B == 0
          R[R.size()-1-L]  +=  Weight * (this->F1 - this->F0) / sA;
        }
      }
    }
  }
}


// B*f+A segment, absolute error
void  CLSLP::Seg::Gen_LinAbs_Segment( vector<complex<double>>&  Q,
                                      vector<complex<double>>&  R,
                                      const double              Weight  )
{
  const double  MinQR_Denom = 1e-6;
  const double  MidPoint = .5 * ( Q.size() - 1 );
        double  Phase0 = 0;
        double  Phase1 = 0;
        double  Denom  = 0;
```

```
// Put in standard form
const double  sB  = (this->A1 - this->A0) / (this->F1 - this->F0);
const double  sA  = this->A0 - sB * this->F0;

// Form The Q Toeplitz Matrix (Vector Here)
for( auto L=0U ; L < Q.size() ; ++L )
{
  Denom = -TPI * L;

  // See if we can assume a zero denominator
  if( abs(Denom) >= MinQR_Denom )
  { // Denominator not zero
    Phase0 = TPI * this->F0 * L;
    Phase1 = TPI * this->F1 * L;

    // Update Q
    Q[L] += Weight * ( complex<double>( cos(Phase1), sin(Phase1) ) -
                       complex<double>( cos(Phase0), sin(Phase0) ) ) /
                       complex<double>( 0.,  -Denom  );
  }
  else
  { // Denominator zero
    // Update Q
    Q[L] += Weight * (this->F1 - this->F0);
  }
}

// Form The R Vector
for( auto L=0U ; L < R.size() ; ++L )
{
  Denom = 4. * PI2 * pow( (double) L - MidPoint, 2 );

  if( abs(Denom) >= MinQR_Denom )
  { // Denominator not zero
    // Update R
    const complex<double>  A1( 0.,  L*TPI*(sB*this->F1 + sA) );
    const complex<double>  A0( 0.,  L*TPI*(sB*this->F0 + sA) );
    const complex<double>  B1( sB, -TPI*MidPoint*(sB*this->F1 + sA) );
    const complex<double>  B0( sB, -TPI*MidPoint*(sB*this->F0 + sA) );

    Phase0 = -TPI * F0 * ( L - MidPoint );
    Phase1 = -TPI * F1 * ( L - MidPoint );

    const complex<double>  X= ((A1 + B1) * complex<double>( cos(Phase1), sin(Phase1) )) -
```

```
                                   ((A0 + B0) * complex<double>( cos(Phase0), sin(Phase0) )) ;

        // Update R
        R[R.size()-1-L] += Weight * X / Denom;
    }
     else
     { // Denominator zero
       // Update R
       R[R.size()-1-L] += Weight * ( ( .5*sB*this->F1*this->F1 + sA*this->F1 ) -
                                     ( .5*sB*this->F0*this->F0 + sA*this->F0 ) );
    }
   }
}


// Must Have At Least 2 Dimensions!!!!!
bool  CLSLP::Levenson( vector<complex<double>>& pR,
                       vector<complex<double>>& g,
                       vector<complex<double>>& X   )
{
  complex<double>          Sum, Delta, Alpha, V;
  vector<complex<double>>  R(X.size());
  vector<complex<double>>  Y(X.size());
  vector<complex<double>>  Z(X.size());
  const double             MinDivisor = 1e-12;

  // Size and data restrictions
  if(pR.size() < 2)   return  false;
  if(g.size() < 2)    return  false;
  if(X.size() < 2)    return  false;
  if( norm(pR[0]) < (MinDivisor*MinDivisor) )  return  false;

  X[0] = g[0]/pR[0];
  Y[0] = -pR[1]/pR[0];
  R[0] = pR[1];

  // Over The Dimensions
  for( auto K=1U ; K < X.size() ; ++K )
  {
    // Calculate Delta
    Sum   = this->InnerProdC(0,K,R,Y);
    Delta = Sum + pR[0];

    if( norm(Delta) < (MinDivisor*MinDivisor) )  return  false;
```

```cpp
    // Calculate V
    Sum = this->InnerProdR(0,K,R,X);
    V   = (g[K]-Sum)/Delta;

    // Update the Z and U vectors
    for( auto L=0U ; L<K ; L++ )
    {
      X[L] += V * conj(Y[K-1-L]);
    }

    X[K] = V;

    if( (K+1) < X.size() )
    {
      // Calculate Alpha
      Sum   = this->InnerProdR(0,K,R,Y);
      Alpha = -(pR[K+1]+Sum)/Delta;

      // Update the Z and U vectors
      for( auto L=0U ; L<K ; ++L )
      {
        Z[L] = Y[L] + Alpha * conj(Y[K-1-L]);
      }

      for( auto L=0U ; L<K ; ++L )
      {
        Y[L] = Z[L];
      }

      Y[K] = Alpha;
      R[K] = pR[K+1];
    }
  }

  // Good return
  return  true;
}


// Linear system solution via Levenson's recursion
complex<double>  CLSLP::InnerProdC( const uint32_t              Start,
                                    const uint32_t              StopPlus1,
                                    const vector<complex<double>>&  InData1,
```

```
                                  const vector<complex<double>>&  InData2   ) const
{
 complex<double>  R = 0;

 for( uint32_t M=Start ; M < StopPlus1 ; ++M )
 {
   R += InData1[M] * conj(InData2[M]);
 }

 return  R;
}


complex<double>  CLSLP::InnerProdR( const uint32_t                Start,
                                    const uint32_t                StopPlus1,
                                    const vector<complex<double>>&  InData1,
                                    const vector<complex<double>>&  InData2   ) const
{
 complex<double>  R = 0;

 for( uint32_t M=Start ; M < StopPlus1 ; ++M )
 {
   R += InData1[M] * InData2[StopPlus1-1U-M];
 }

 return  R;
}
```