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

March 11, 2025

William Earl Jones

http://www.wejc.com || wejc@wejc.com

Last Updated 03/13/25 at 20:03

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 are 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 on a standard desktop PC.

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 first described in reference [**1**]. CLSLP filters are a minimum weighted square-error type and possess linear-phase by design.

If we can generate a linear-phase FIR filter we can always create another one with exactly the same amplitude response by adding a constant offset to it's phase. As long as the phase slope in frequency is correct, the filter will have exactly the sample amplitude response as the original filter as well as possessing linear phase. Using this argument, conjugate-symmetric, conjugate-asymmetric, and hybrid linear-phase FIR filter coefficient configurations are all possible using the CLSLP algorithm. Member function *GenFilter* has a third optional

parameter which is *RotAngRad*, or the rotation angle in radians. Setting *RotAngRad* to *0* results in the generation of a conjugate-symmetric linear-phase FIR filter which is the default. Setting *RotAngRad* to *M_PI/2* results in the generation of a conjugate-asymmetric linear-phase FIR filter again with the same amplitude response. Setting *RotAngRad* to a value between *0* and *M_PI/2* generates a hybrid filter somewhere between a symmetric and asymmetric linear-phase FIR filter. *RotAngRad* is periodic with period π. The amplitude response of the filter will be identical though for any value of *RotAngRad (*radians). *RotAngRad* defaults to 0 or a conjugate-symmetric filter.

CLSLP filters are generated without a Fourier Transform so the fidelity of the filter coefficients is better than what would be expected: *-150 dB* stop-band in Figure 3.1 below. The CLSLP algorithm is also computationally efficient, and physically small with a memory footprint of *11 KBytes* on a desktop PC. These attributes make CLSLP ideal for embedded signal processing tasks.

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.

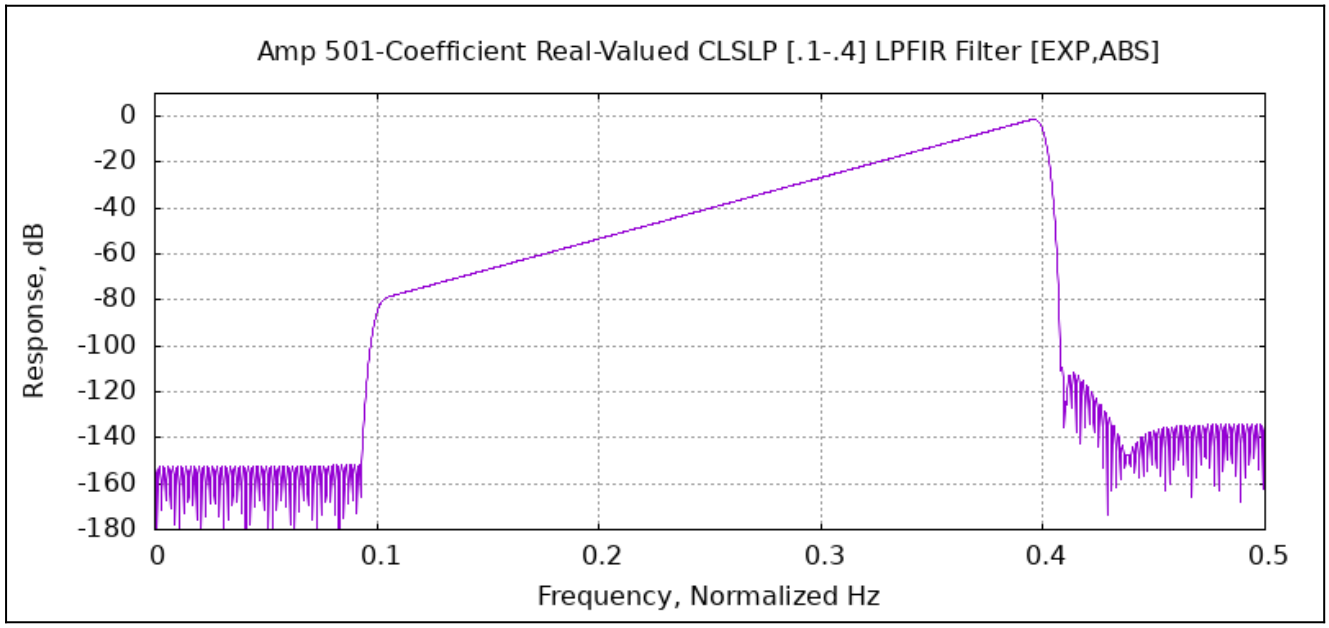

Figure 3.1: 501-Coefficient Real-Valued CLSLP [.1-.4] LPFIR Filter [EXP,ABS].

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 and is used if this parameter is omitted. A segment weight of *2* doubles the fit error in that frequency interval. Increasing the frequency segment weight will improve the fit in that segment at the expense of fit in other frequency segments. 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**
- **CLEAN**
- **BUILD_AND_RUN**
- **GENERATE_PLOTS**

The *Start.cpp* module is the C++ *main* program. As a local convention we use S*tart* 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

The following commands are used to clean, build, run, then generate graphic plot files. The sub-directory *build* is usually the CMAKE build sub-directory. 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 (optimized compilation)*, is sufficient for basic SIMD support on CPUs that support it. Intel and AMD PC CPUs normally do support it.

```
// 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.GenerateFilter( 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.

```
#! /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_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 *1.57 ms* 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.

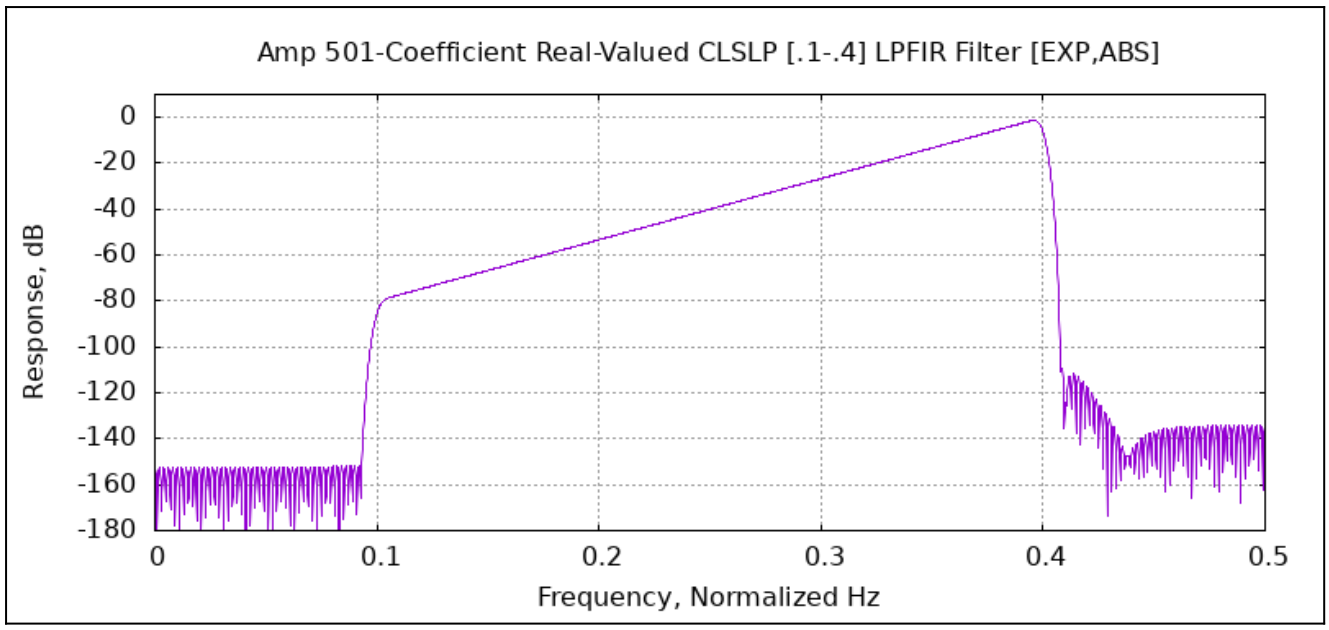

Figure 4.1: 501-Coefficient Real-Valued CLSLP [.1-.4] LPFIR Filter [EXP,ABS].

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.GenerateFilter( 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 *1.56 ms* 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.

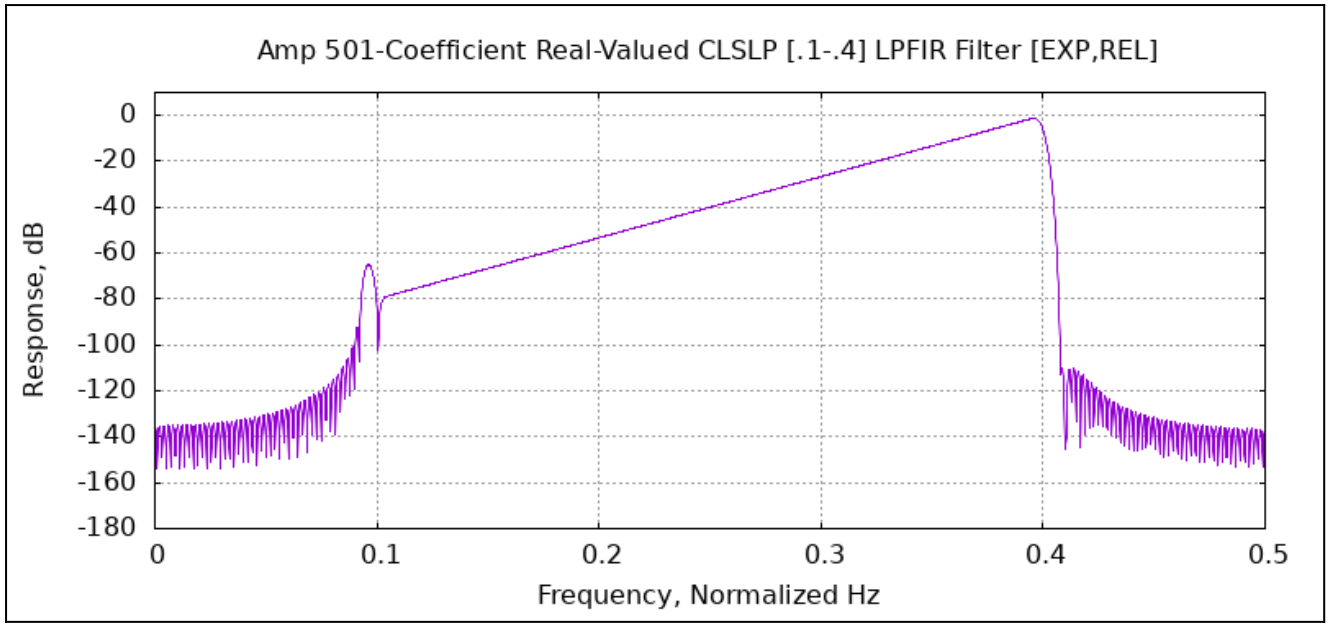

Figure 4.2: Amplitude: 501-Coefficient Real-Valued CLSLP [.1-.4] LPFIR Filter [EXP,REL].

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

```
// 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.GenerateFilter( 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 *3.52 ms* or *7.02 µs* per coefficient. The linear slope appears as a logarithmic curve.

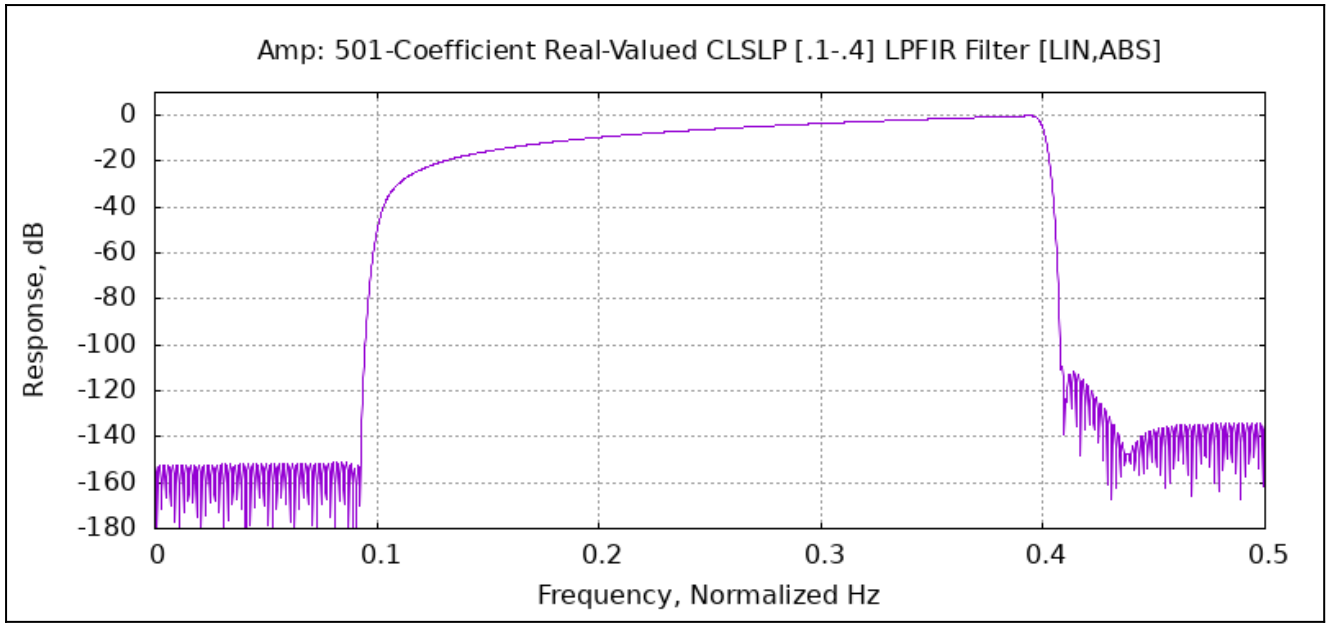

Figure 4.3: Amplitude: 501-Coefficient Real-Valued CLSLP [.1-.4] LPFIR Filter [LIN,ABS].

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

```
// 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.GenerateFilter( 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 *1.64 ms* 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 *1.64 ms* or *3.26 µs* per coefficient. The linear slope appears as a logarithmic curve.

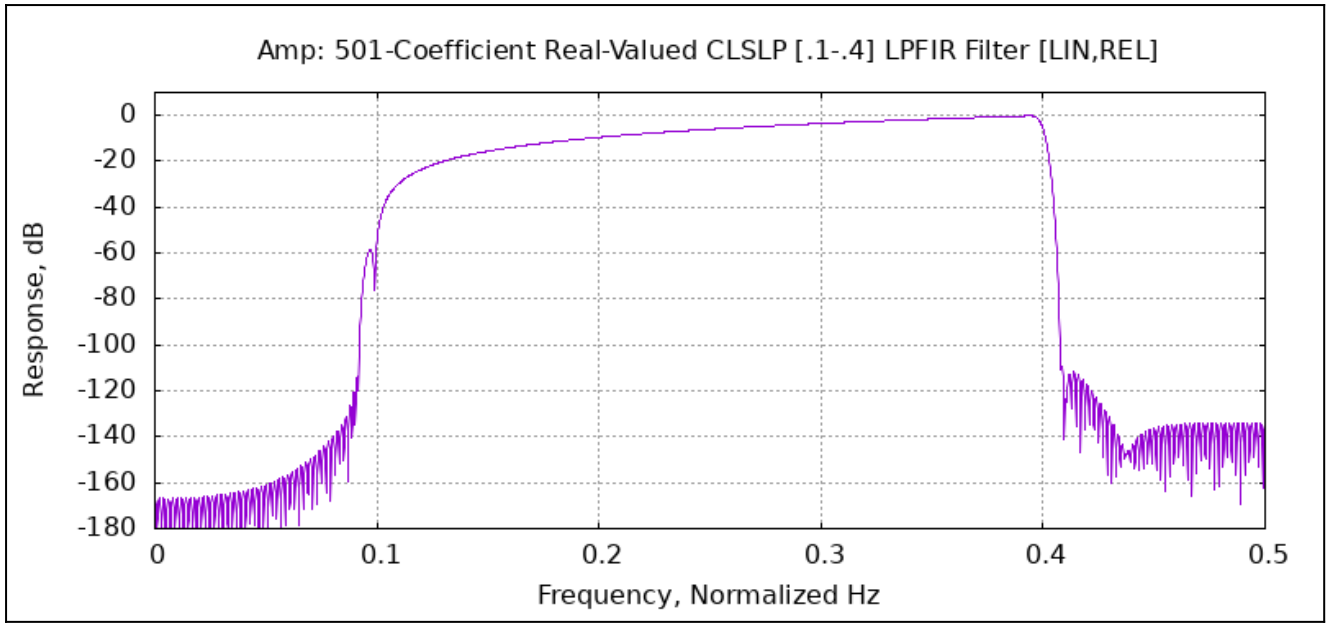

Figure 4.4: Amplitude: 501-Coefficient Real-Valued CLSLP [.1-.4] LPFIR Filter [LIN,REL].

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

```
// 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.GenerateFilter( 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.GenerateFilter( NumFiltCoefs, OutCoefs );
```

This filter synthesis takes *5.54 ms* or *5.54 µs* per coefficient on a standard PC. The resultant CLSLP filter's amplitude response is given in Figure 4.5 below.

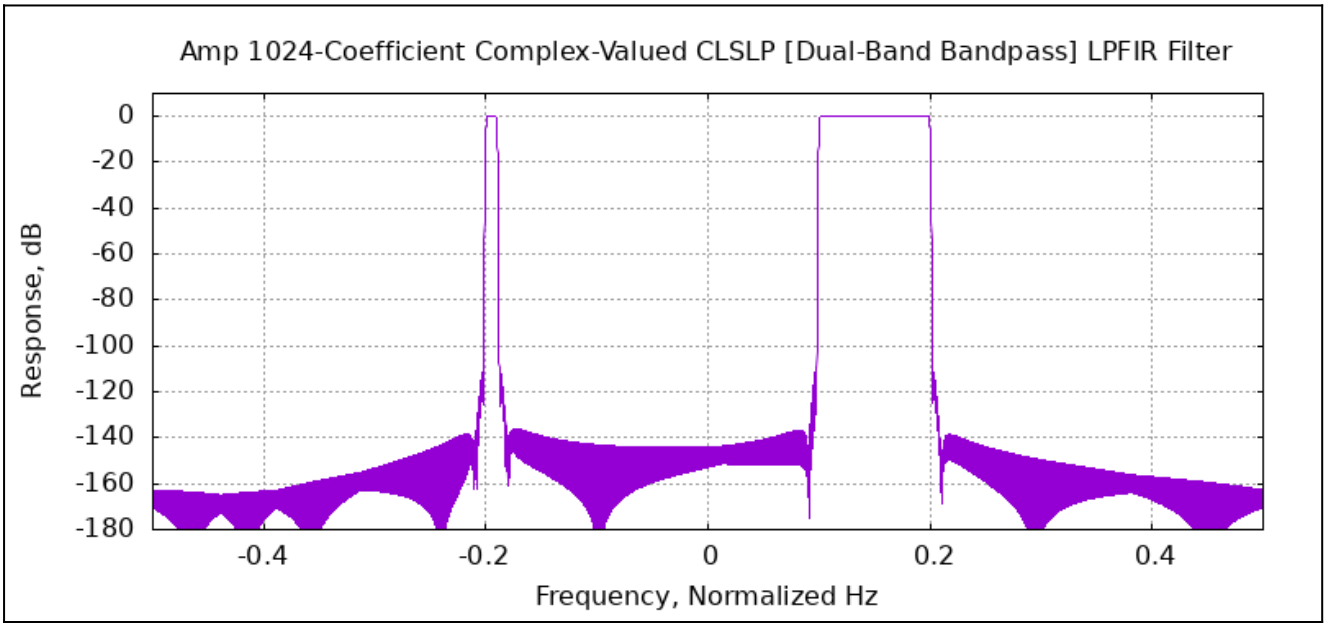

Figure 4.5: Amplitude: Ex5 1001-Coefficient Complex-Value CLSLP LPFIR [EXP,ABS].

4.2.2. Example Filter 6

A linear-phase FIR filter with *2049* complex-valued coefficients is synthesized using the CLSLP algorithm. We are testing the complex-coefficient mode so amplitude plots will be displayed from *0* to *1* 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 = 2049;
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( CLSLP::eExp_AbsErr, -0.50, -0.20, 0.00, 0.00, 1.0 );
Filt.Add( CLSLP::eExp_AbsErr, -0.20, -0.19, 1.00, 1.00, 1.0 );
Filt.Add( CLSLP::eExp_AbsErr, -0.19, 0.10, 0.00, 0.00, 1.0 );
Filt.Add( CLSLP::eExp_AbsErr, 0.10, 0.20, 1.00, 1.00, 1.0 );
Filt.Add( CLSLP::eExp_AbsErr, 0.20, 0.40, 0.00, 0.00, 1.0 );
Filt.Add( CLSLP::eExp_AbsErr, 0.40, 0.50, 0.00, 0.00, 1.0 );

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

This filter synthesis takes *22.38 ms* or *10.92 µs* per coefficient on a standard PC. The resultant CLSLP filter's amplitude response is given in Figure 4.5 above.

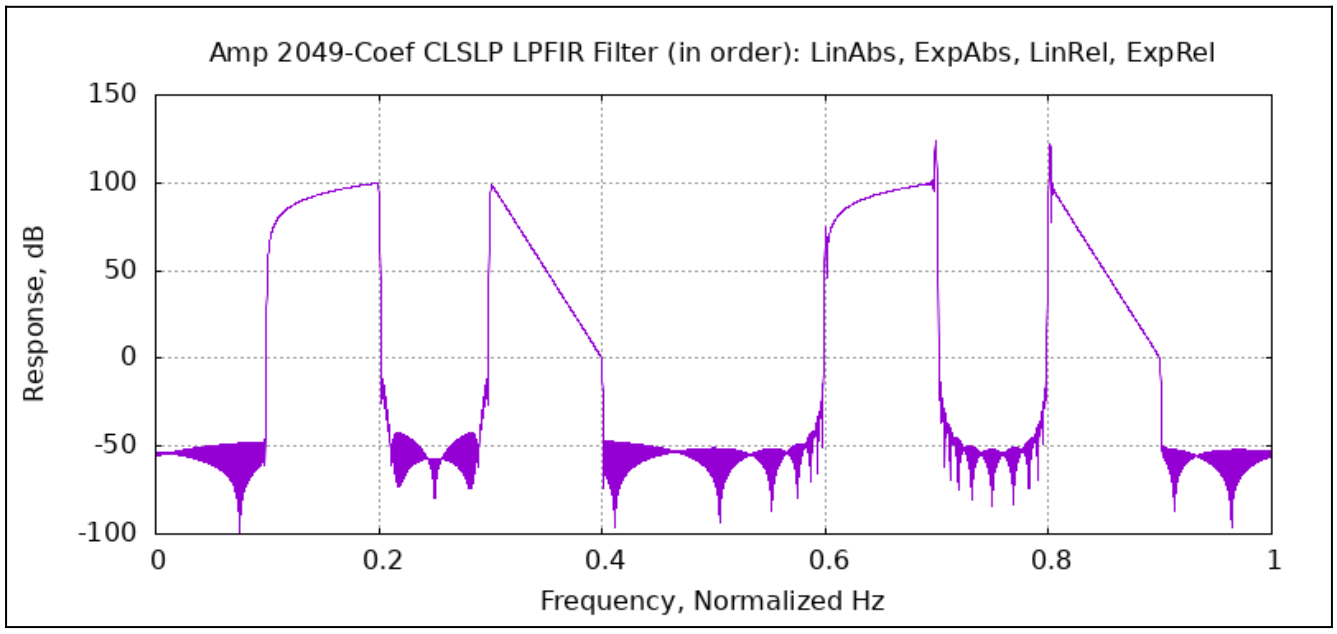

Figure 4.6: Amplitude: Ex6 2047-Coefficient Complex-Value CLSLP LPFIR [EXP,ABS].

This filter synthesis takes *23.39 ms* or *11.43 µs* per coefficient on a standard PC. The resultant CLSLP filter's amplitude response is given in Figure 4.6 above.

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.
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).
AlignedMemAlloc.h	SSE and AVX C++ vector allocator for faster execution on SIMD machines.
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.