General Purpose FIR Filters

Paul Byrne 9/11/21

This repository contains general purpose FIR filters, implemented in both C and C++. The filters utilise both circular buffer arrays and STL vectors. Also included are two filter design applications in Python 3.8 + Qt5 user interface. One for High Pass/Low Pass, the other for Band Pass/Band Stop. They include a basic selection of windowing options.

Table of Contents

1 Principles Using Circular Buffer Arrays

In those filters which use circular buffer arrays, array indexing is used to ensure correct application of the FIR filter taps.

Example;

In this test example a small 11 tap filter was applied to input data. To make examination easier the input data is incremented by 1 on each data sample. Below is when the buffer data index is 10, which represents to most recent data sample. It can be seen that the FIR taps are applied in the correct sequence, tap 1 to the most recent data and tap 11 to the oldest.

Data index is 10

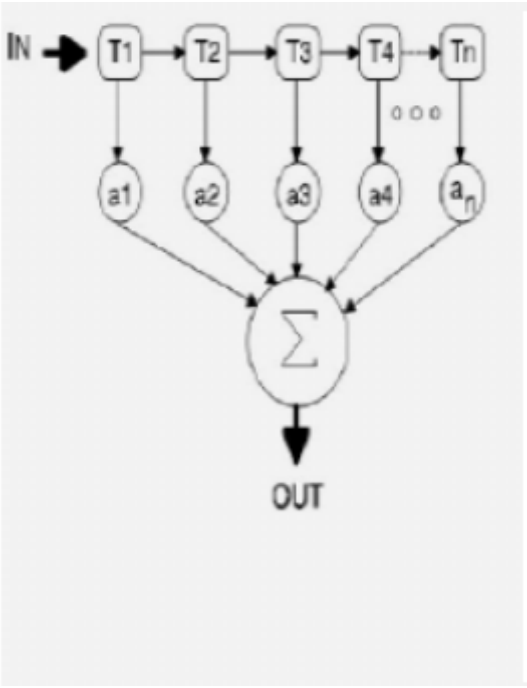
Data Buffer	Data Value	Filter Tap No.	Filter Tap Value
00	11.000000	11	-5.215538e-03
01	12.000000	10	-8.040162e-03
02	13.000000	9	1.335863e-02
03	14.000000	8	1.057387e-01
04	15.000000	7	2.405481e-01
05	16.000000	6	3.072205e-01
06	17.000000	5	2.405481e-01
07	18.000000	4	1.057387e-01
08	19.000000	3	1.335863e-02
09	20.000000	2	-8.040162e-03
10	21.000000	1	-5.215538e-03

On the next sample the circular data buffer now rolls over and the data buffer index is now 00, making index 00 the most recent data sample.

Data index is 00

Data Buffer	Data Value	Filter Tap No.	Filter Tap Value
00	22.000000	1	-5.215538e-03
01	12.000000	11	-5.215538e-03
02	13.000000	10	-8.040162e-03
03	14.000000	9	1.335863e-02
04	15.000000	8	1.057387e-01
05	16.000000	7	2.405481e-01
06	17.000000	6	3.072205e-01
07	18.000000	5	2.405481e-01
08	19.000000	4	1.057387e-01
09	20.000000	3	1.335863e-02
10	21.000000	2	-8.040162e-03

It can be seen that the FIR tap indexing is adjusted to match the data buffer index as the circular buffer rolls over.



2 Testing

All the filters were tested using FIR coefficients generated using Python applications (see FIR Filter Design in section 3).

The test code used a simulated test signal which the FIR filters processed. Filters for low-pass, high-pass, band-pass and band-stop were applied, all with a hamming window.

The simulated signal used for testing was 10Hz primary modulated by 100Hz and 200Hz both at 10% of the amplitude of the primary, with a sample rate of 1000Hz.

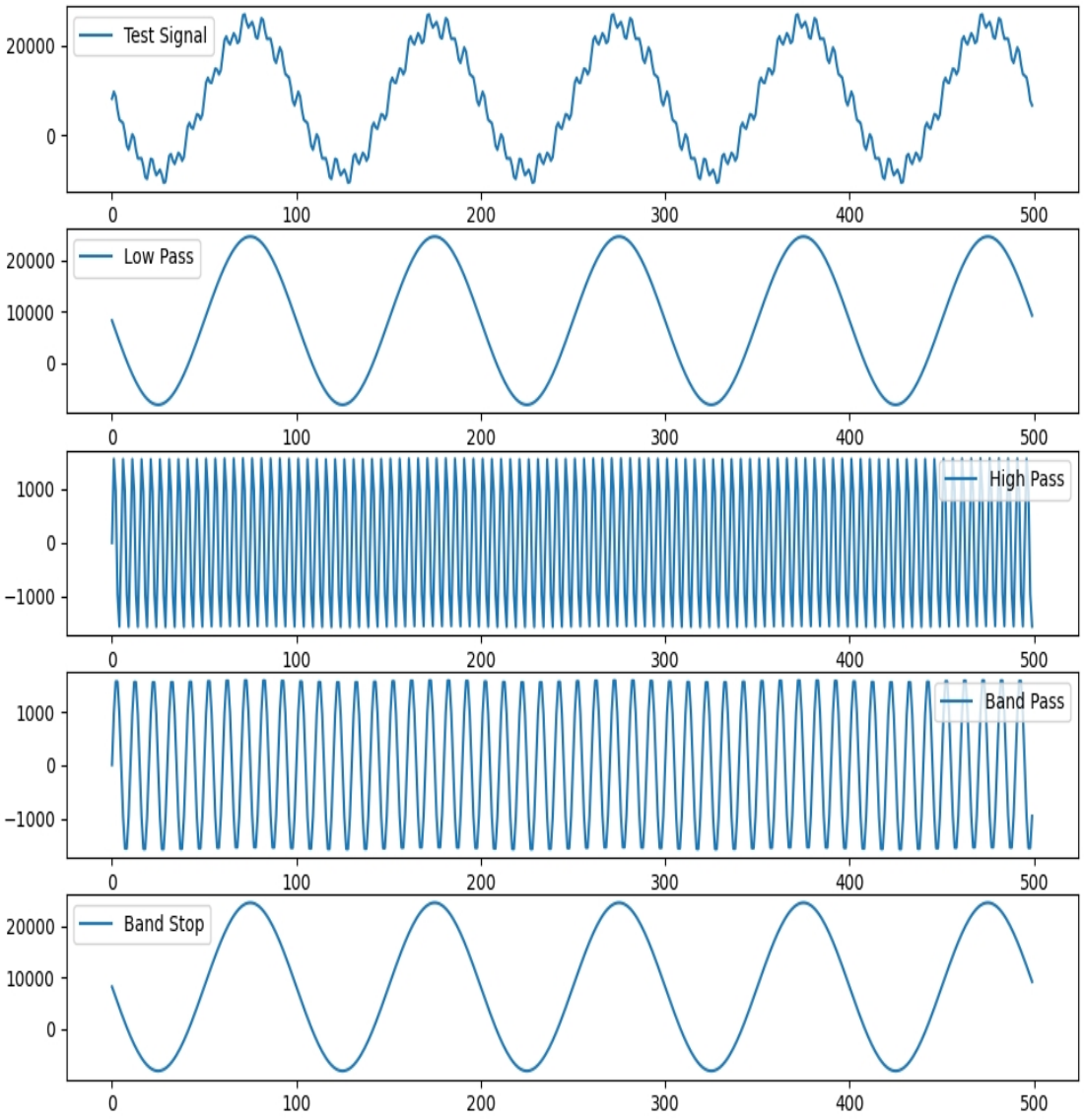
Low pass filter; cut off freq= 50.0Hz transition_freq=20Hz window= hamming

High Pass; cut off freq= 150.0Hz transition_freq=20Hz window= hamming

Band Pass; start freq= 50.0Hz Stop Freq=150.0Hz transition_freq=20Hz window= hamming

Band Stop; start freq= 50.0Hz Stop Freq=300.0Hz transition_freq=20Hz window= hamming

The results obtained from all the filter versions is shown below.



3 In Use

Loading Filter Coefficients

Both the C and C++ versions have functions for loading the filter coefficients via a text file containing the coefficients, or to directly input an array containing the FIR coefficients.

When FIR coefficients are loaded from a text file, the first line is expected to be the FIR filter specification, followed by the coefficients This is the format generated by the Python applications. For example the first three lines of a bandpass filter is shown below;

```
sample rate= 1000.0Hz  start freq= 50.0Hz Stop Freq=150.0Hz  transition_freq=20Hz  window= hamming   Filter= bandpass
-2.436477280702506e-18
-0.000129154528152310
```

The first line is not operated on by the code, it is only there to assist users and is ignored. However if coefficients are loaded from another source, it will need to be modified so that a dummy line is placed at the top of the file before the filter coefficients as in the example above.

Filtering

All versions of the FIR filters have a function or method called **Filter**. Here the input value is passed by reference, the filtered output result is obtained also by reference. The folder **testing** has examples of their use.

Speed

The filters were testing by filtering a million samples using a 201 tap filter. The test code was built with the -O3 optimisation. No CPU optimisation was applied. The code was run on a 10 year old desktop;

C code filter 4.5 Million samples/sec.

C++ code filter using arrays 4.4 Million samples/sec.

C++ code filter using STL vectors 2.0 Million samples/sec.

While not CPU optimised these are useful speeds, although the vector implementation is the slowest.

4 Code

C Code

The C code consists of two files **fir.c** and **fir.h**. In use the **cfilt_filesetup** function is called with the filename containing the filter coefficients. An alternative function **cfilt_setup** can be used when an array of the filter coefficients as available. Filtering is applied by calling **cfilt_filter** function were the input and output data are passed by reference. ***Note*** that when exiting the code the **cfilt_reset_memory** function **must** be called to deallocate memory.

C++ Code Using Arrays

The C++ code consists of two files **fir.cpp** and **fir.hpp**. In use the **CLPFilter::FileSetup** method is called with the name of the file containing the filter coefficients. An alternative method **CLPFilter::Setup** can be used when an array of the filter coefficients as available. Filtering is applied by calling **void CLPFilter::Filter** method were the input and output data are passed by reference. Memory is deallocated automatically in the destructor.

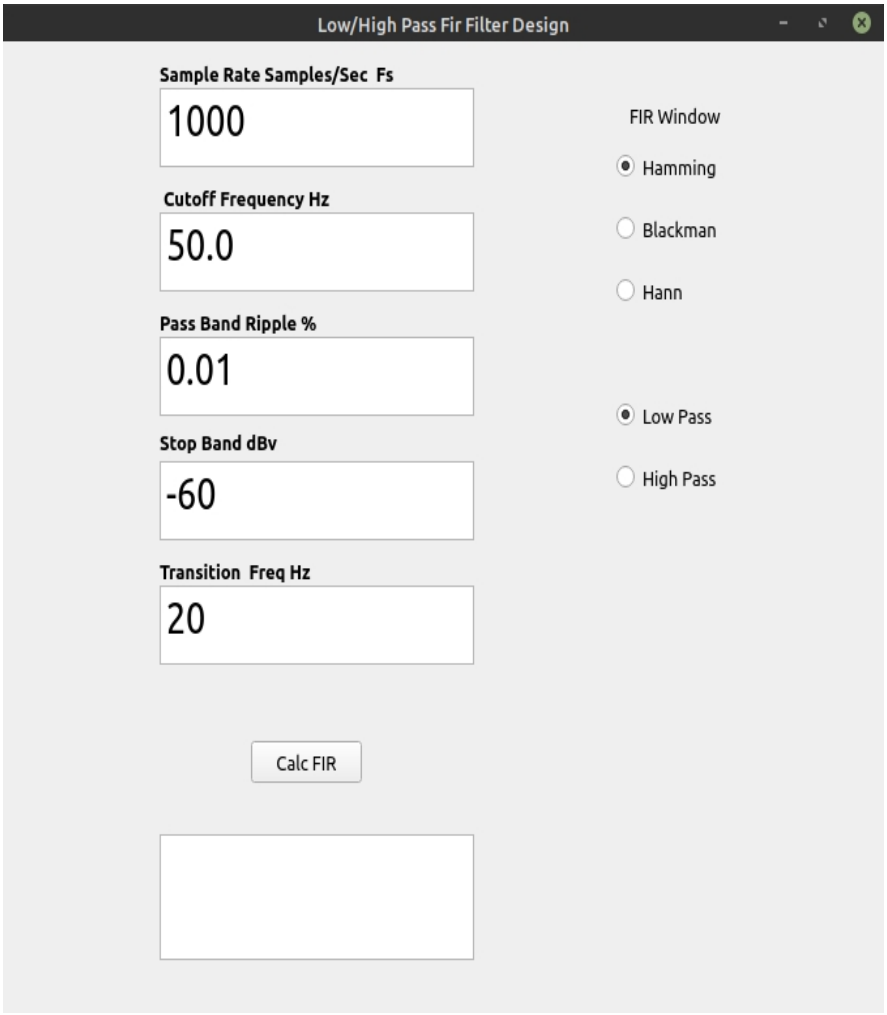
C++ Code Using Vectors

The C++ code consists of two files **firvect.cpp** and **firvect.hpp**. In use the **CLPFilter::FileSetup** method is called with the name of the file containing the filter coefficients. An alternative method **CLPFilter::Setup** can be used when an array of the filter coefficients as available. Filtering is applied by calling **void CLPFilter::Filter** method were the input and output data are passed by reference. Memory used by the vectors is automatically deallocated..

FIR Filter Design

Two filter design applications were produced, one for high/low pass filters, the second for bandpass/ bandstop. The applications were written in Python using Qt5 for the user interface. The filter coefficient file will be stored in the same folder that the application is run in. To run these applications Qt5 must be installed.

The Low/High Pass application is run from command line by **./fircalc1.py** and the Bandpass/Bandstop **./fircalcband.py**. When run the initial default values displayed are those used in some of the testing (following pages).


Low/High Pass Fir Filter Design
Sample Rate Samples/Sec Fs
1000
Cutoff Frequency Hz
50.0
Pass Band Ripple %
0.01
Stop Band dBv
-60
Transition Freq Hz
20
Calc FIR
FIR Window
Hamming
Blackman
Hann
Low Pass
High Pass

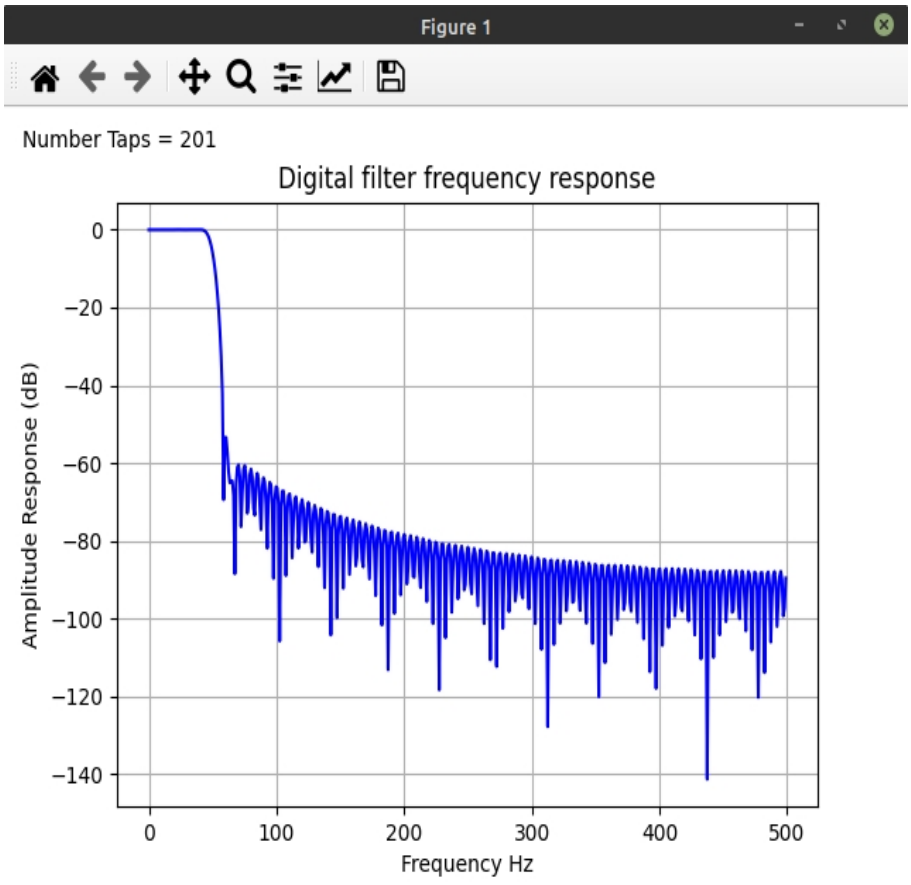
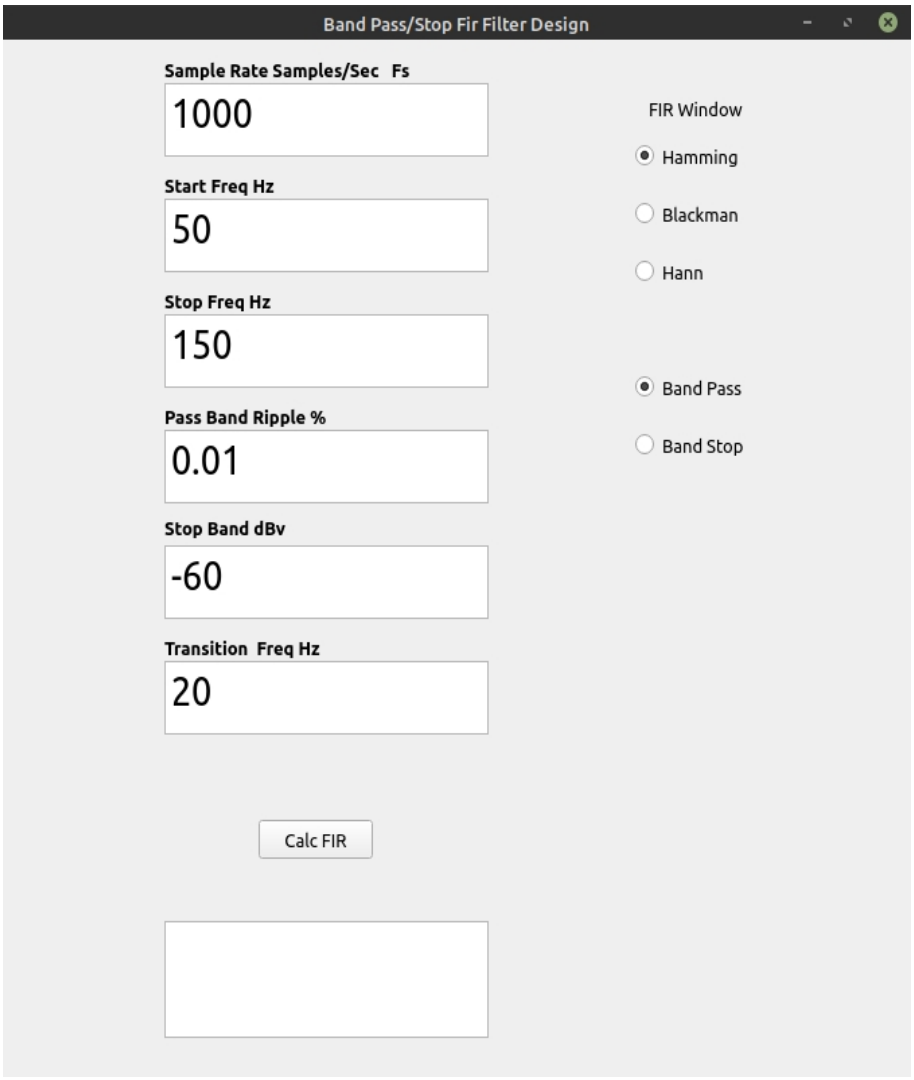


Figure 1
Number Taps = 201
Digital filter frequency response
Amplitude Response (dB)
0
−20
−40
−60
−80
−100
−120
−140
0
100
200
300
400
500
Frequency Hz


Band Pass/Stop Fir Filter Design
Sample Rate Samples/Sec Fs
1000
Start Freq Hz
50
Stop Freq Hz
150
Pass Band Ripple %
0.01
Stop Band dBv
-60
Transition Freq Hz
20
Calc FIR
FIR Window
Hamming
Blackman
Hann
Band Pass
Band Stop

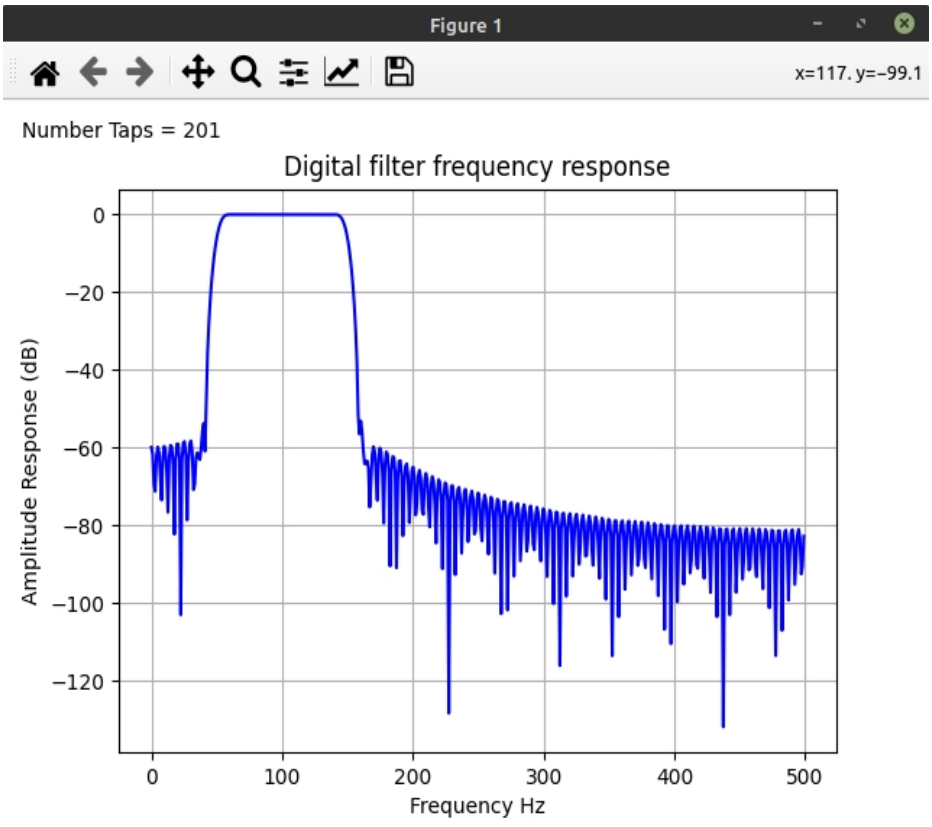


Figure 1
x=117. y=−99.1
Number Taps = 201
Digital filter frequency response
Amplitude Response (dB)
0
−20
−40
−60
−80
−100
−120
0
100
200
300
400
500
Frequency Hz

5 Appendix

Folder Contents

doc	This document **firfilter.pdf**
filtercode	Code for C and C++ versions of fir filter, in **csource**, **cppsource** and **cppsource_vector** subfolders respectively.
firsdesign	Python scripts and Qt5 ui for low pass/ high pass, and band pass/band stop filter design applications. The **lohi** subfolder contains the low/high pass application **py** and **ui** files, the **bandpass** subfolder the bandpass/bandstop p**y** and **ui** application files.
testing	Code used in testing filters. Contains **filterc, filtercpp** and **filtercpp_vector** subfolders. Each as relevant make file for generating test code, plus original filter files used.

Paul Byrne 9/11/21