Iowa Hills Software Digital and Analog Filters
Example C Code for FIR and IIR Filters Home

FIR and IIR Source Code Kit.

The source code in this kit will synthesize Windowed FIR filters, Parks McClellan FIR filters, and IIR filters using the bilinear transform. The code will synthesize low pass, high pass, band pass, and notch filters, up to 10 poles.

The coefficients required to synthesize IIR Butterworth, Chebyshev, Inverse Chebyshev, Bessel, Gauss, Adjustable Gauss, and Elliptic polynomials are also included.

Code Kit Download (zip) Updated 12/5/14

Kit Contents For Filters

FIR	IIR
Rectangular Window Impulse Response	Bilinear Transform Synthesis
Parks McClellan	s Plane Filter Coefficients
Impulse Windows (Kaiser, Sinc, etc)	Form 1 Biquad Implementation Code
Implementation Code	Frequency Analysis code.
Frequency Analysis code.	Quartic & Cubic Root Solvers
Frequency Correction Code	

Kit Contents for Spectral Analysis

An FFT Algorithm.	Two DFT Algorithms
Goertzel Algorithm.	Spectral Analysis Windows. (Hanning, Gauss, Flattop, etc.)

The following is a subset of the code in the kit.

IIR Filters

See this page for IIR Filter Design Equations and C Code. It gives the equations used to generate IIR filters from the s domain coefficients of analog filters using the Bilinear Transform. The design equations for low pass, high pass, band pass, and notch filters are given.

FFT Algorithm and Spectral Analysis Windows

See this page for an FFT Algorithm in C. The only difficult part of writing an FFT algorithm is generating the various array indexes, the rest of the code is trivial. The associated Butterfly Chart is also given as well as ways to optimize an FFT for speed. The need for windows is discussed, and an explanation is given for when an FFT's output should be scaled by 1/N.

Parks McClellan C++ Source Code

Here are two versions of the Parks McClellan algorithm translated from Fortran to C. Both files are essentially straight c, and should compile with few, if any changes. We started with the original Parks McClellan Fortran code given in this Wikipedia article.

In the first file we tried to keep the changes to a minimum. We then used the SciPy code to check ours against. As we noted in the file, there are small differences between our translation and the SciPy code that you may want to look at. Search for "SciPy" in our file for the notes.

We then decided to make extensive changes to the code in an effort to improve it's readability. We eliminated 62 of the original 69 goto statements, added two functions, deleted the Hilbert and Differentiator code, and deleted or renamed numerous variables. We had hoped to eliminate all the goto statements, but there are 4 in the Remez function so intertwined that we couldn't see a way to eliminate them. We tested our new code by comparing its output to the coefficients from our original translation on more than 1000 different filters. We also compared some filters to the filters generated by other programs on the web.

We also included some code in the second program to calculate and range check the band edges. This makes for a simpler function call, but more importantly, restricting the band edge values helps to ensure convergence.

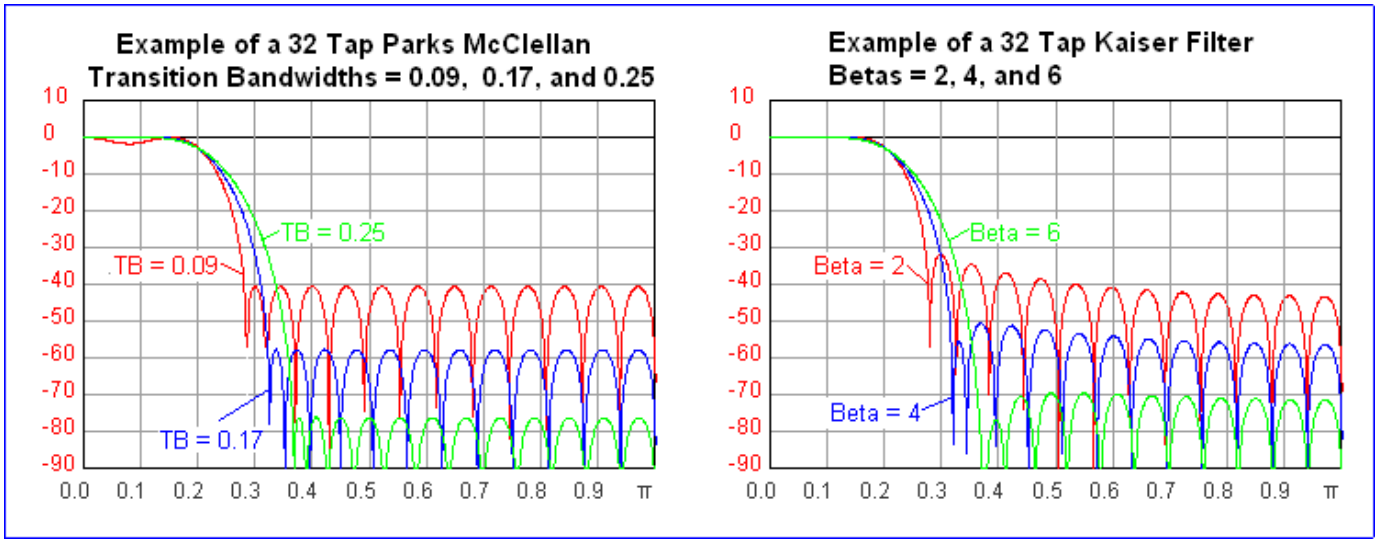
The 1st program's input: "64,1,3,0.0,0.1,0.2,0.35,0.425,0.5,0.0,1.0,0.0,10.0,1.0,10.0"

The 2nd program's call: NewParksMcClellan(NumTaps, OmegaC, BW, ParksWidth, PassType);

We removed the Hilbert Transform code from the second version because Parks McClellan doesn't generate a good Hilbert transform. Hilbert filters are used in applications that require little or no ripple. For example, when used to generate single sideband, the ripple must typically be less than 0.01 dB in order to achieve the desired sideband suppression. So it is counter productive to synthesize the filter with an algorithm that creates ripple by design. Thus, its a matter of using the right tool for the job, and the Fourier Transform is a much better tool for synthesizing Hilbert filters. See our Hilbert Filters page for examples.

Parks McClellan C++	This code has the minimum number of changes to get to a C translation and is comparable to the SciPy code.
New Parks McClellan C++	This code has the changes described above. It does more error checking and has a more friendly call.

On the left is an example of a 32 tap Parks McClellan filter for 3 transition bandwidth settings. On the right is a 32 Tap Kaiser Filter (generated by the window code given below).



C Code for Windowed FIR Filters.

If you are writing a program to generate FIR filters, or want to implement an algorithm in MathCAD, then by all means, start with a windowed filter. These are very good filters and can be generated with less than 10 lines of code.

This Windowed FIR Filter C Code has two parts, the first is the calculation of the impulse response for a rectangular window (low pass, high pass, band pass, or notch). Then a window (Kaiser, Hanning, etc) is applied to the impulse response. There are several windows to choose from, but we recommend you start with the Kaiser because you can adjust its transition bandwidth and sidelobe levels. The Sinc window is also adjustable and also very good.

An example of a 32 tap Kaiser filter generated by this code is shown above.

The file contains code for these windows.

Hanning	Hamming	Blackman
Blackman Harris	Blackman Nuttall	Nuttall
Kaiser	Kaiser Bessel	Trapezoid
Sinc	Flattop	Tukey
Sine	Gauss	

For an implementation of the algorithms given here, see our free FIR Filter Designer . It is also capable of synthesizing filters from adjustable non rectangular windows such as the Raised Cosine, and polynomials such as the Bessel. The Parks McClellan algorithm is also implemented.

The code below is for implementing and analyzing FIR and IIR filters. This code was clipped from our FIR and IIR filter design programs, but clipping code from a program isn't without its hazards. There may be an omission, such as an undeclared variable, but the essence of the code (the technique) should be clear.

IIR Filter Implementation Code

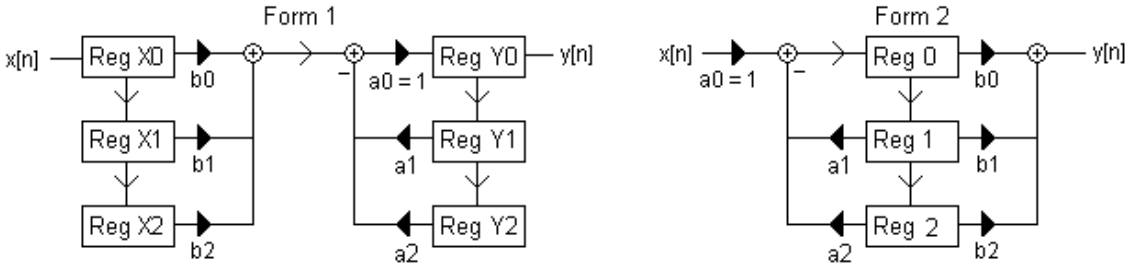
Because of the numerical difficulties associated with IIR filters, many different ways have been developed to implement them. Highly selective IIR filters are particularly susceptible to register overflow and round off errors which can destroy the filter's performance. We give the code for the four implementations discussed in most textbooks. We list them here in order of their numerical performance.

1. Form 1 Biquad (Best)
2. Form 2 Biquad
3. Form 1 Nth Order Poly
4. Form 2 Nth Order Poly (Worst, not recommended)

Code for Biquad Implementations (2nd Order Sections).

These structures implement an IIR filter as a series of 2nd order sections. These are preferred over an Nth order implementation for numerical reasons. The Form 1 implementation is the better of the two. Note the minus sign next to a0. You may need to make a sign change depending on how your IIR coefficients are generated.

IIR 2nd Order Implementations (Biquads)



$$H(z) := \frac{b_2 \cdot z^{-2} + b_1 \cdot z^{-1} + b_0}{a_2 \cdot z^{-2} + a_1 \cdot z^{-1} + a_0}$$

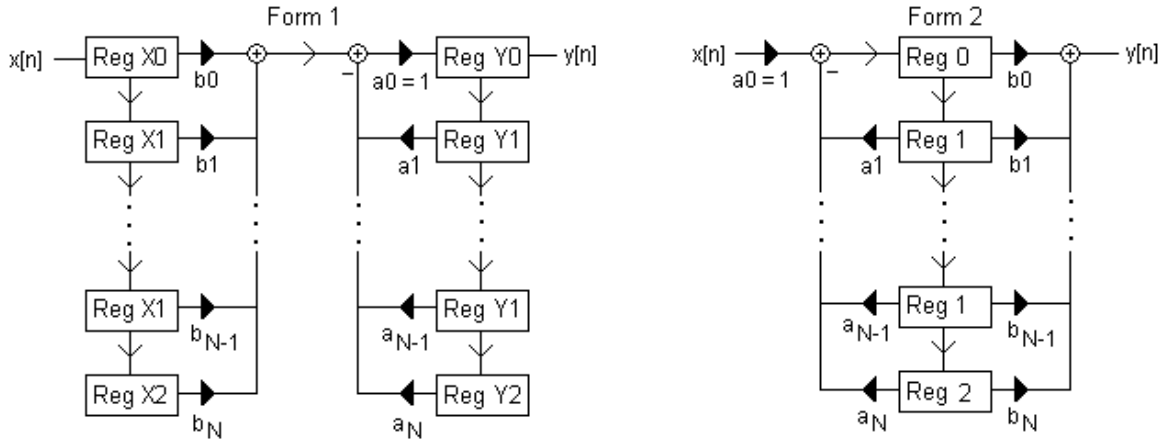
IIR Filter Biquad Implementation Code

Code for Nth Order Polynomial Implementations.

These structures implement an IIR filter as a single Nth order polynomial. These implementations are **not** well suited for a fixed point processor because of the peak math values generated, especially Form 2. This code is a bit simpler than the biquad code however.

Form 2 is quite similar to the way FIR filters are typically implemented, but FIR filters don't have the same numerical problems because they don't have any feedback (the denominator).

IIR Nth Order Implementations



$$H(z) = \frac{b_n \cdot z^{-n} + b_{n-1} \cdot z^{-(n-1)} + \cdots + b_1 \cdot z^{-1} + b_0}{a_n \cdot z^{-n} + a_{n-1} \cdot z^{-(n-1)} + \cdots + a_1 \cdot z^{-1} + a_0}$$

IIR Filter Nth Order Implementation Code

Code to Implement FIR filters.

There are two methods given here for implementing an FIR filter. The first is a straight forward implementation of an FIR flowchart that uses a loop to shift the delay register values. The second implements the same flowchart, but rotates register indexes rather than the register values. It is more efficient, but not quite as easy to follow.
FIR Filter Floating Pt. Implementation Code

Calculating a Digital Filter's Frequency Response.

The frequency response of an FIR filter is usually obtained by taking the FFT of the coefficients. Simply zero pad the coefficients to give a convenient FFT length. Take note however that most FFT algorithms scale a forward transform by 1/N (the FFT length), but this scaling isn't appropriate when doing the transform of an impulse response. So, depending on your FFT, you may need to multiply the output by N to get the correct gain.

We can also do a Discrete Fourier Transform of the coefficients. The best reason for using this approach is that it allows us to choose the evaluation frequencies, rather than be restricted to the FFT's bin frequencies.

Another approach is to use the Goertzel algorithm. This algorithm is typically used to detect a single tone at a given frequency, but if you only need the magnitude response of your filter at a single frequency, it is simpler and faster than a single frequency DFT, but doesn't provide phase information.

This file has two DFT implementations and a Goertzel implementation.
Frequency Analysis Code for FIR Filters

As with an FIR filter, the easiest way to analyze an IIR filter's frequency response is to run an impulse through the filter and FFT the output. This approach becomes impractical however if the filter has a very narrow bandwidth. Then a very large FFT is required in order to get good resolution in the filter's pass band, which can be cumbersome.

Another approach is to do two FFT's on the Nth order numerator and denominator coefficients, and divide the results, but this has the same problem just mentioned for highly selective filters.

This method does a DFT on the filter's second order coefficients. Its main advantage is that it allows you to restrict your analysis frequencies to the band of interest.
Frequency Analysis Code for IIR Filters

Numerical Algorithms

akiti.ca is an excellent source for a Polynomial Root Finder in C. We use this code in our filter design programs.

Its origin is the Fortran code on this site: http://www.netlib.org/toms/ (affiliated with The University of Kent).

Another good source for numerical algorithms is Numerical Recipes in C , Cambridge University Press, (requires flash).