MUSC 208 Winter 2014
John Ellinger Carleton College

Lab 17 Filters II

Lab 17 needs to be done on the iMacs.

Five Common Filter Types

Lowpass
A low pass filter allows low frequencies to pass through and attenuates high frequencies.

Highpass
A high pass filter allows high frequencies to pass through and attenuates low frequencies.

Bandpass
A band pass filter allows is a combination of a low pass and a high pass filter. A band pass filter allows frequencies between the start and stop bands to pass through and attenuates frequencies outside those bands.

Bandreject or Notch
A band notch filter attenuates frequencies between the start and stop bands.

Resonant
A resonant filter amplifies a very narrow band of frequencies.

Finite Impulse Response (FIR) Filters

FIR filters operate only on input samples, current and past. Lab16 looked at two simple FIR filters, a weighted Lowpass filter and a weighted Highpass filter.

$$output[n] = a_0 \bullet input[n] \pm a_1 \bullet input[n-1]$$

Adding the two inputs input produced a lowpass filter and subtracting produced a highpass filter. The weights are the values of a0 and a1 which summed to zero.

Infinite Impulse Response (IIR) Filters

IIR filters operate on input and output samples, current and past. Lab 17 will look at two IIR filters: a simple first order IIR lowpass and highpass filter and a second order IIR filter, also known as the biquad filter, that is one of the most versatile filters in digital audio.

Filter Order

The order of a filter is the determined by the number of remembered past inputs and outputs. The filters from Lab16 are first order FIR filters because they only remember one input sample in the past. The biquad filter is a second order IIR filter because it remembers the previous two input samples and previous two output samples.

First Order IIR Filer

This formula is very similar to the Lab16 FIR filter, except the second term is the past output sample instead of the past input sample.

$$output[n] = a_0 \bullet input[n] \pm a_1 \bullet input[n-1], \text{FIR}$$

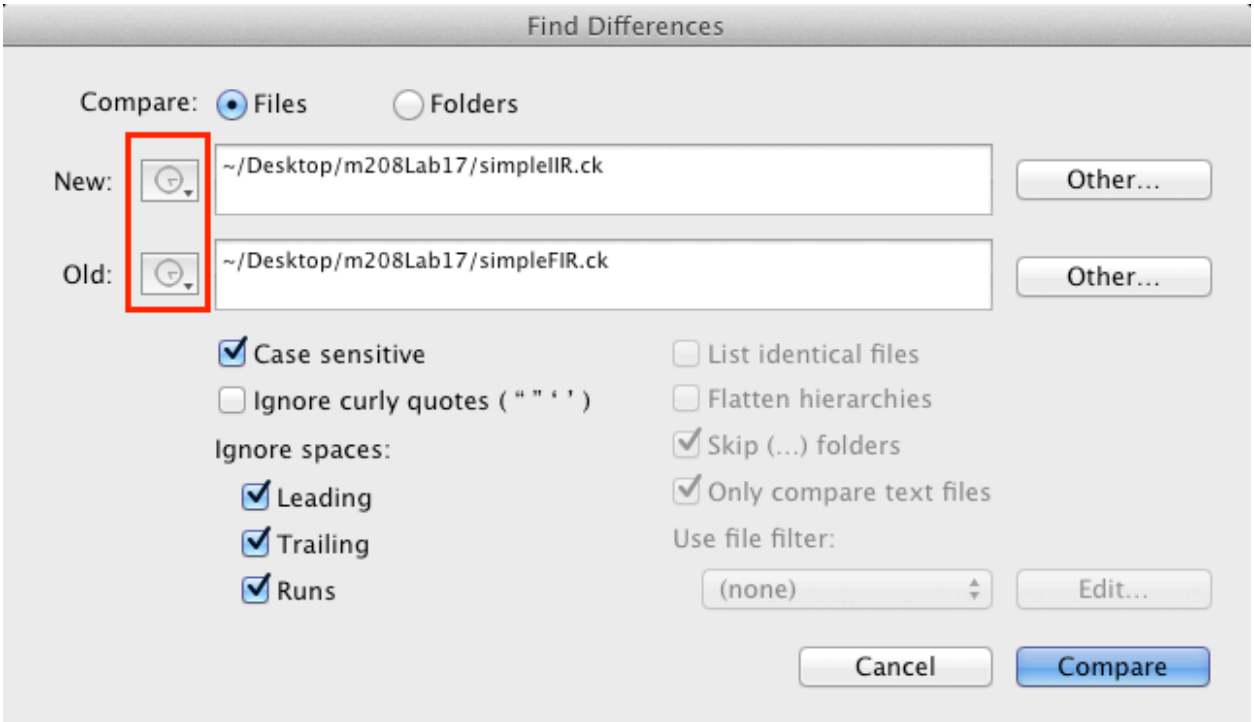
$$output[n] = a_0 \bullet input[n] \pm b_1 \bullet output[n-1], \text{IIR}$$

Download the m208Lab17 folder

The m208Lab17/Lab17_IIR folder contains ready to run Processing and ChucK code to test the first order IIR filter. It was a simple rewrite of the Lab16 code.

Compare Files

Use TextWrangler to see exactly what has changed from the Lab16 FIR code to the Lab17 IIR code. Open the simpleFIR.ck and simpleIIR.ck files in TextWrangler. Choose Find Differences from the Search menu. Click the New and Old popup menus and select the two ChucK files.



Click Compare.

The two source file windows will be displayed side by side along with a third window at the bottom that shows lines that differ. Click on the first difference. Use the arrow keys to see the other differences. Most programmer text editors have a similar file compare tool. There are also several freeware file and folder comparison tools for both Mac and Windows available on the internet.

We're almost ready to test the first order IIR filter but first we have to go through the same audio setup steps we did in Lab 16.

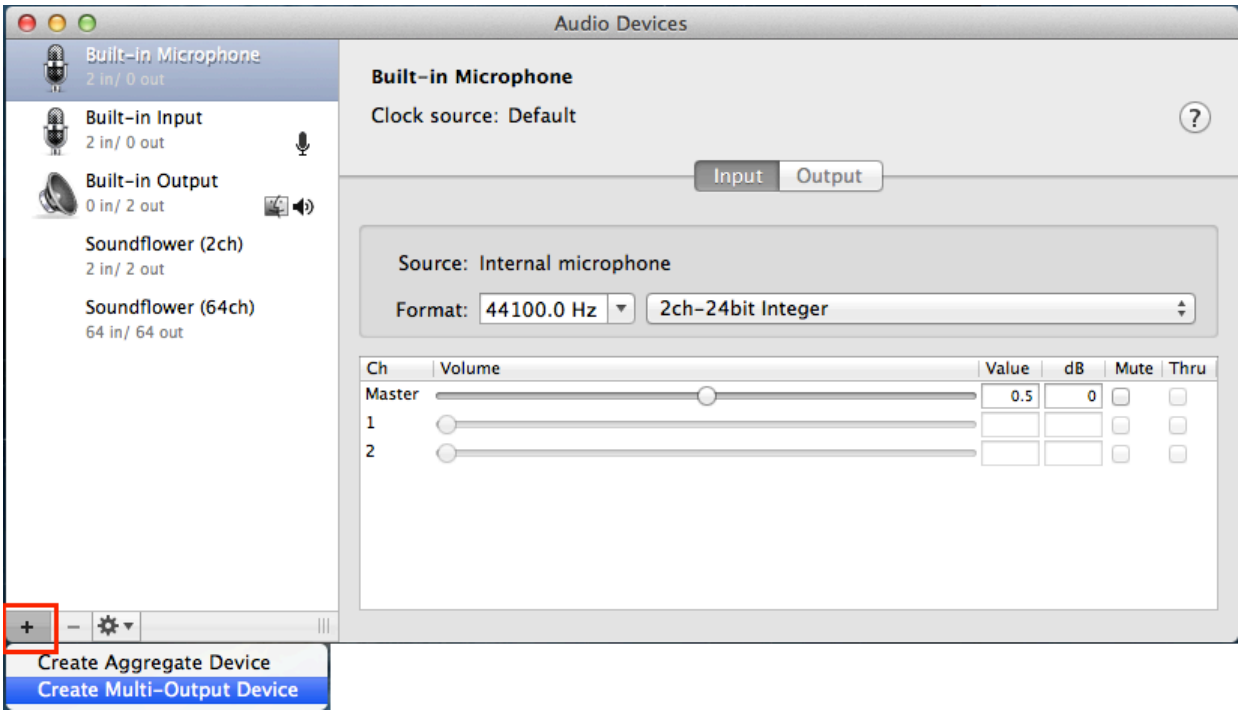
Setup

Lab 17 needs to be done on the iMacs.

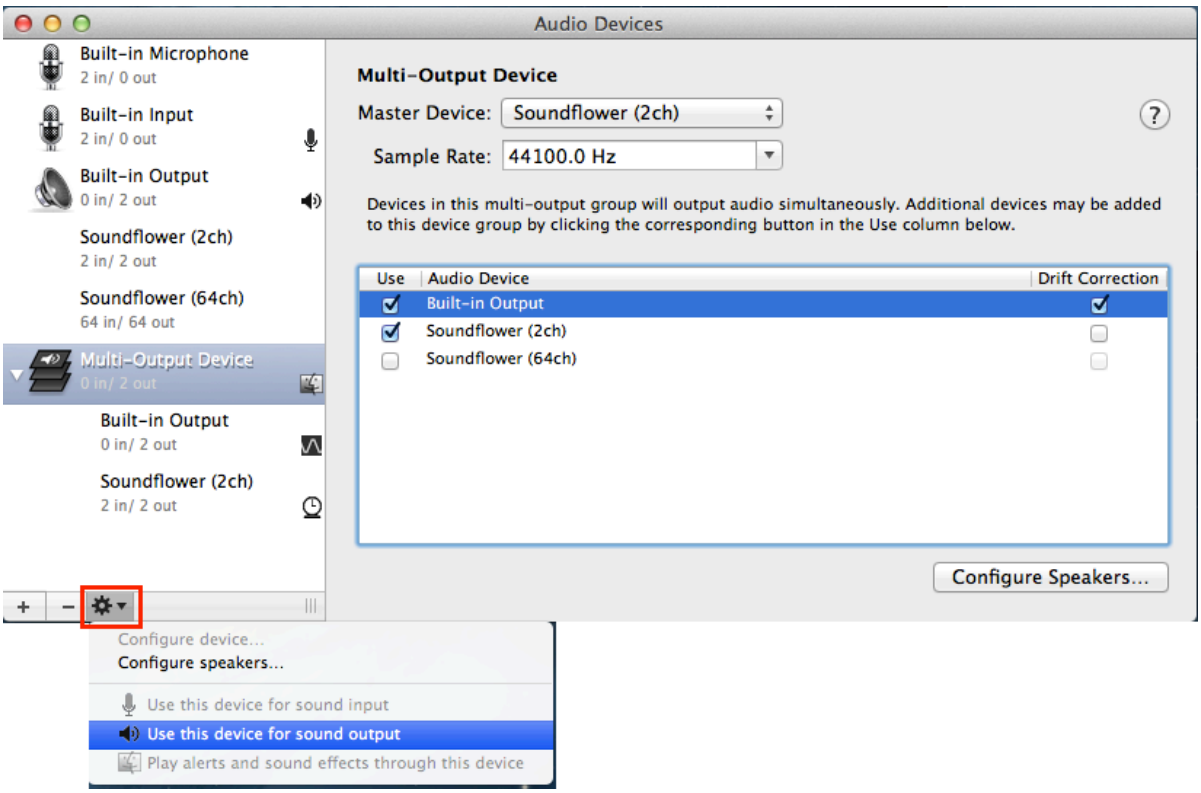
You're going to create a Multiple Output Audio Device that will send ChucK audio through SoundFlower into SignalScope and at the same time send it to the speakers.

Audio MIDI Setup

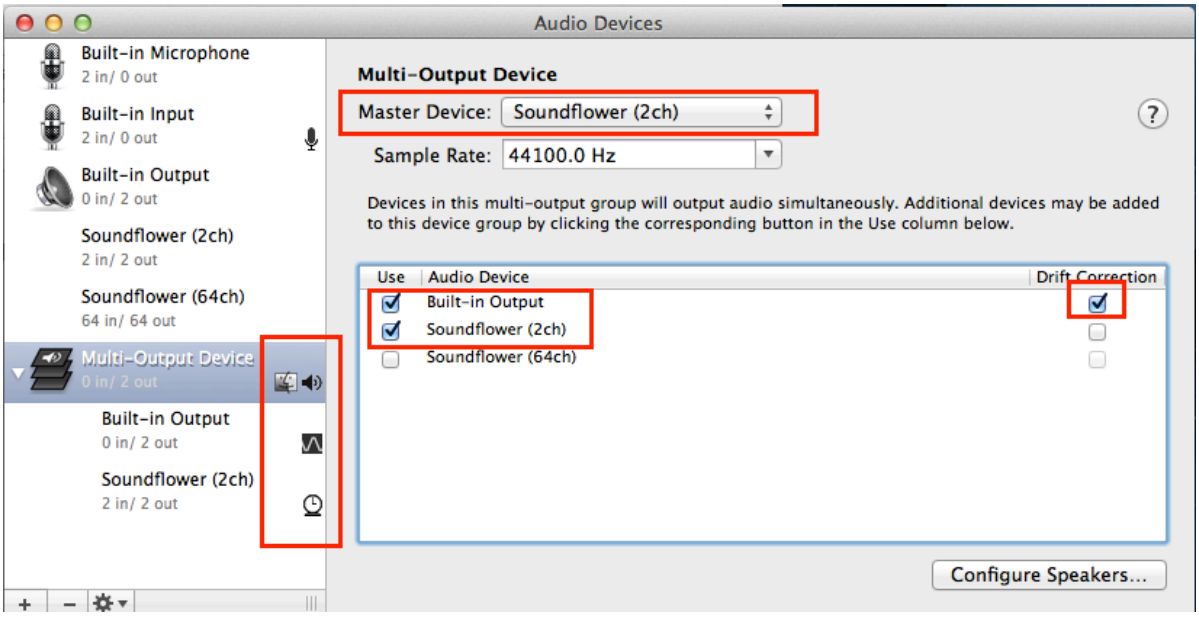
Open /Applications/Utilities/Audio MIDI Setup.



Use the Window menu to display the Audio Devices window. Click the plus sign at the bottom of the Audio Devices window and choose Create Multi-Output Device.

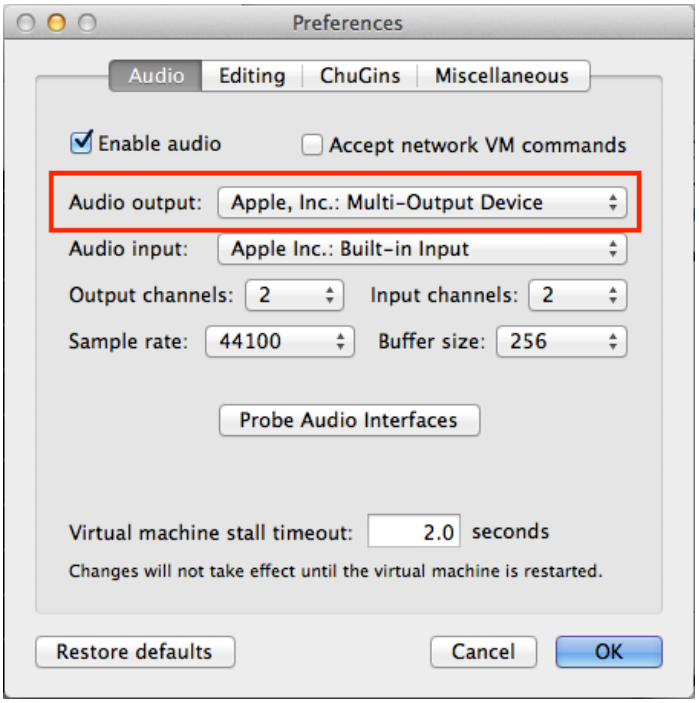


Select the Multi-Output Device in the left panel and click the pop-up menu that looks like a gear and choose Use this device for sound input. Choose Soundflower 2(ch) as the Master device. Check Built-in Output and Soundflower as Audio devices. Check Drift Compensation for Built-in Output. The completed setup should look like this. Quit Audio MIDI Setup.



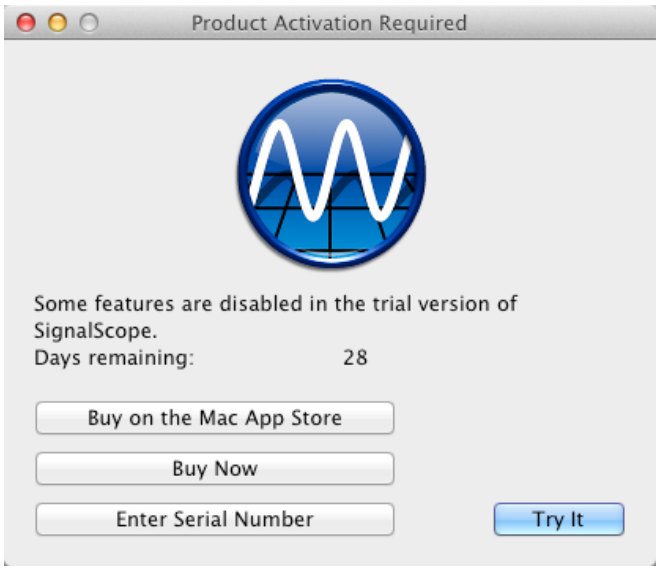
miniAudicle

Open /Applications/miniAudicale and then open the Preferences dialog. Choose the Multi-Ouput Device for Audio output. Quit and restart miniAudicle.

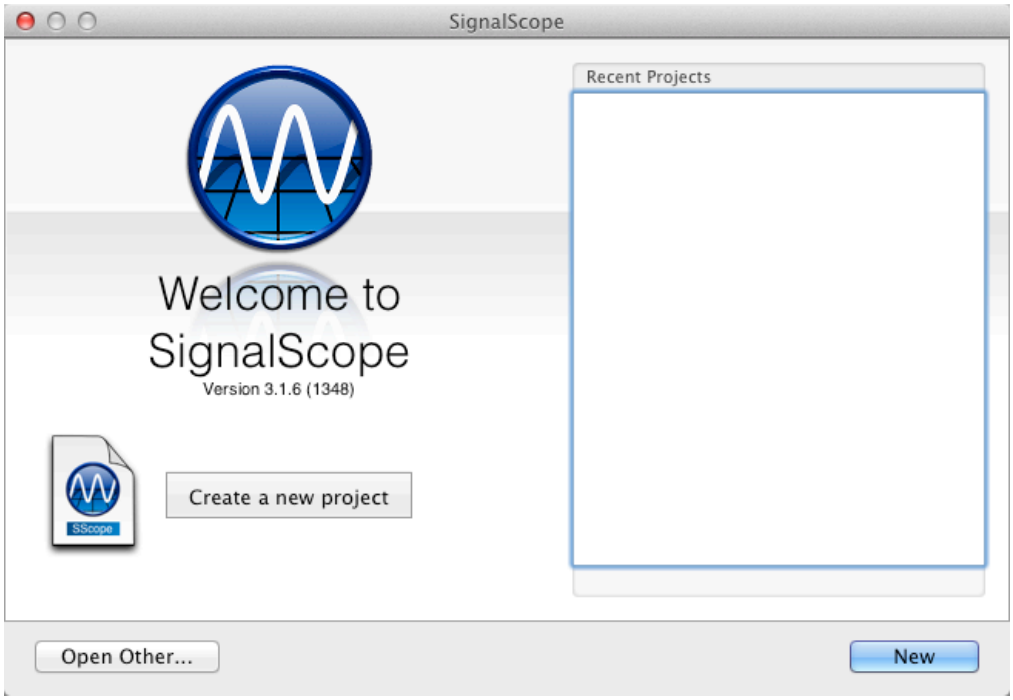


Signalscope

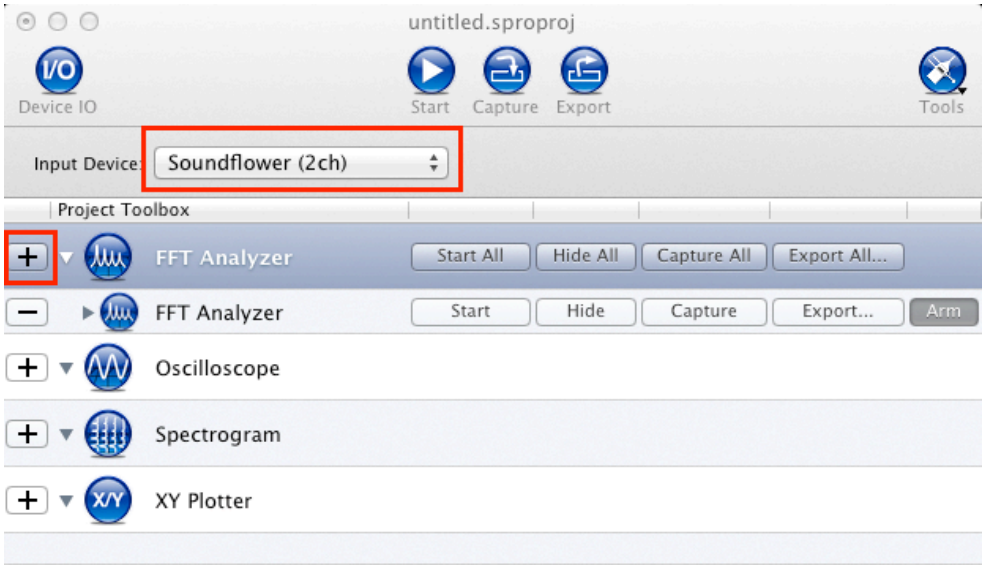
Open /Applications/SignalScope. Click the Try It button



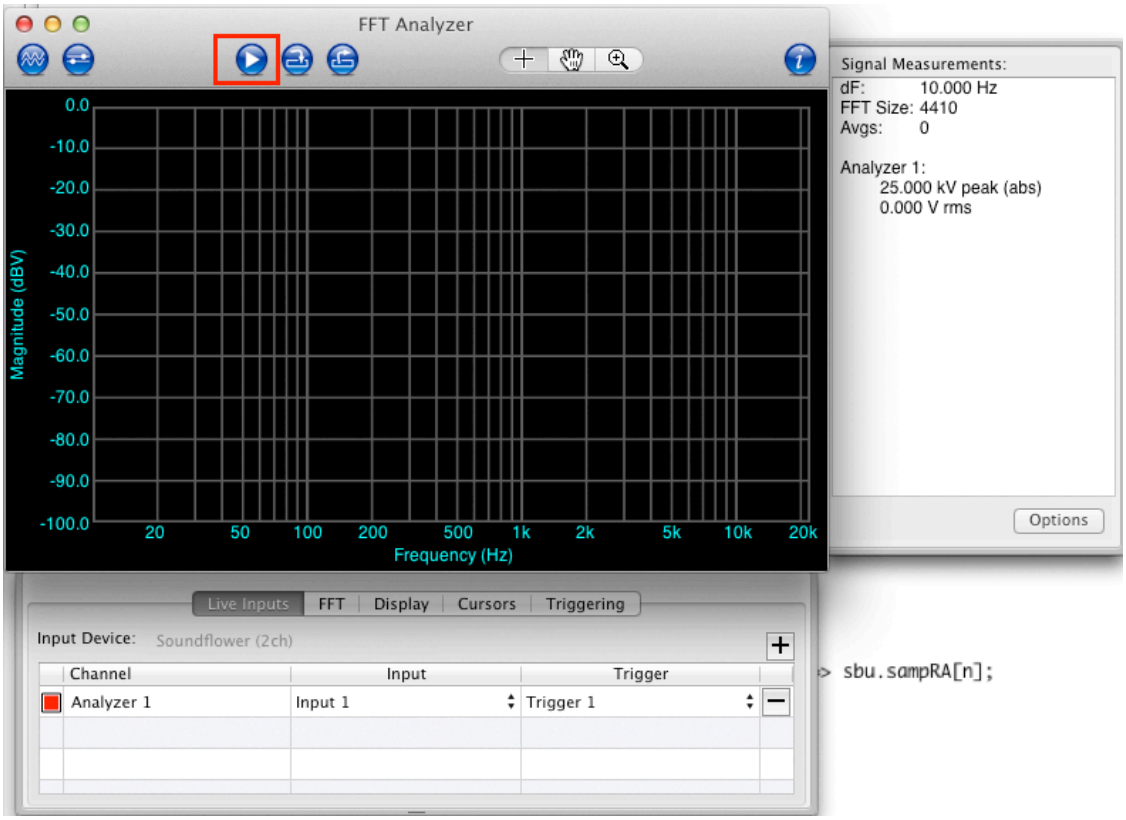
Click the New button



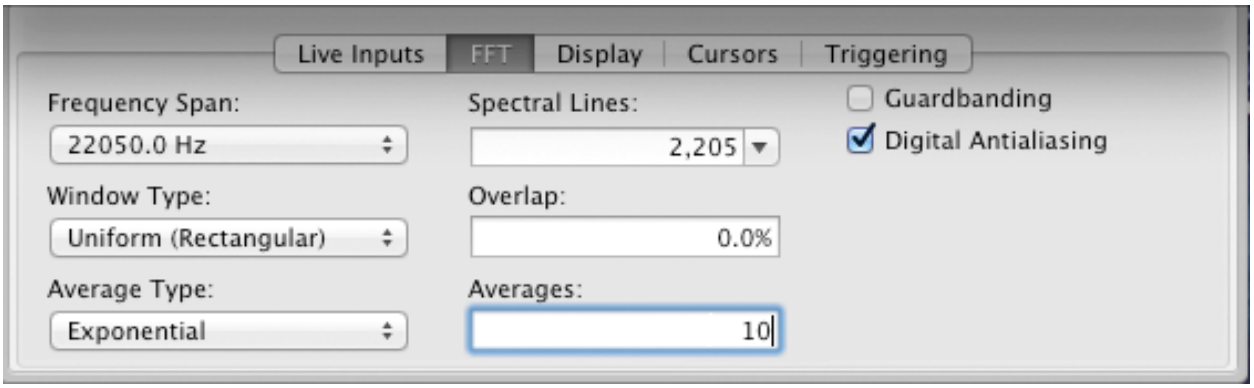
Click the Plus sign to the left of FFT Analyzer and choose Soundflower as the input device.



The FFT Analyzer window will appear.



Click the FFT tab and verify these settings.



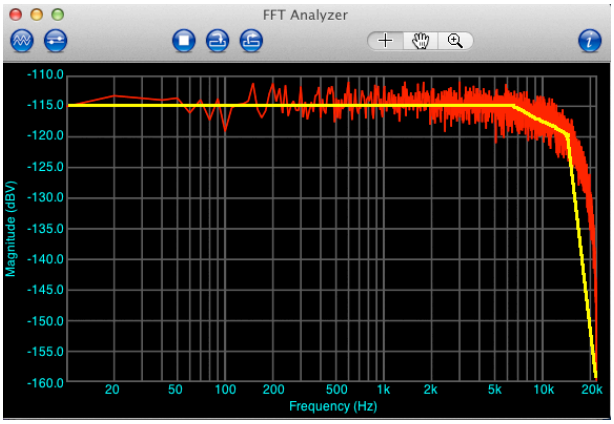
Simple IIR Filter

These ready to run files are found in the m208Lab17/simpleIIR download folder. Run Lab17_IIR.pde in Processing and then run runSimpleIIR.ck in miniAudicle. Experiment with the GUI controls and watch the waveform that appears in the SoundScope FFT Analyzer. The results you hear will sound

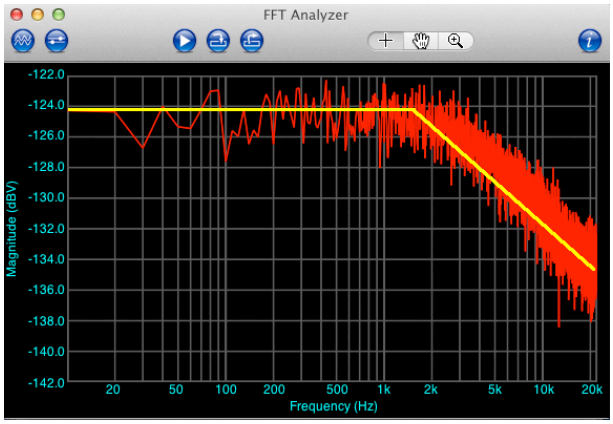
similar to the Lab16 FIR filter, but the IIR filter is subtly different from the FIR filter as these screenshots show. All GUI sliders were set to 0.50. The lowpass IIR filter of Lab17 attenuates high frequencies sooner (at a lower cutoff point) than the FIR filter of Lab16. The IIR highpass filter attenuates low frequencies much longer than the FIR highpass filter. The yellow lines were added to emphasize the filter shape.

Lowpass

Lab 16

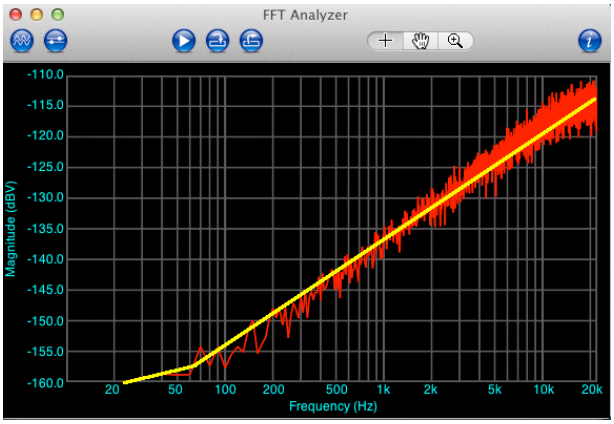


Lab 17

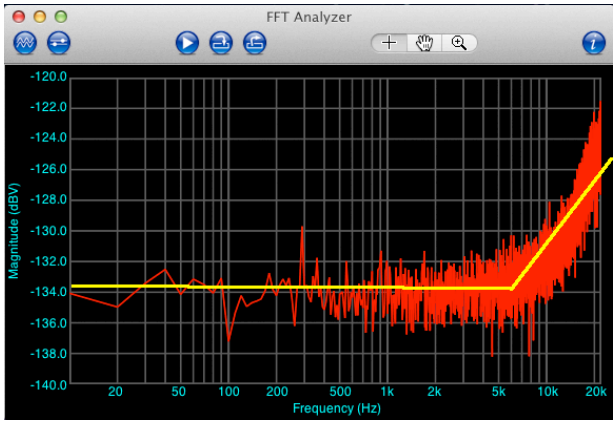


HighPass

Lab 16



Lab 17



The Biquad Filter

In the following formula above z^{-1} is equivalent to x(n-1) and z^{-2} is equivalent to x(n-2). To quote Wikipedia http://en.wikipedia.org/wiki/Digital_biquad_filter:

"In signal processing, a digital biquad filter is a second-order recursive linear filter, containing two poles and two zeros. "Biquad" is an abbreviation of "biquadratic", which refers to the fact that in the Z domain, its transfer function is the ratio of two quadratic functions:

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

High-order recursive filters can be highly sensitive to quantization of their coefficients, and can easily become unstable. This is much less of a problem with first and second-order filters; therefore, higher-order filters are typically implemented as serially-cascaded biquad sections (and a first-order filter if necessary). The two poles of the biquad filter must be inside the unit circle for it to be stable. In general, this is true for all filters i.e. all poles must be inside the unit circle for the filter to be stable."

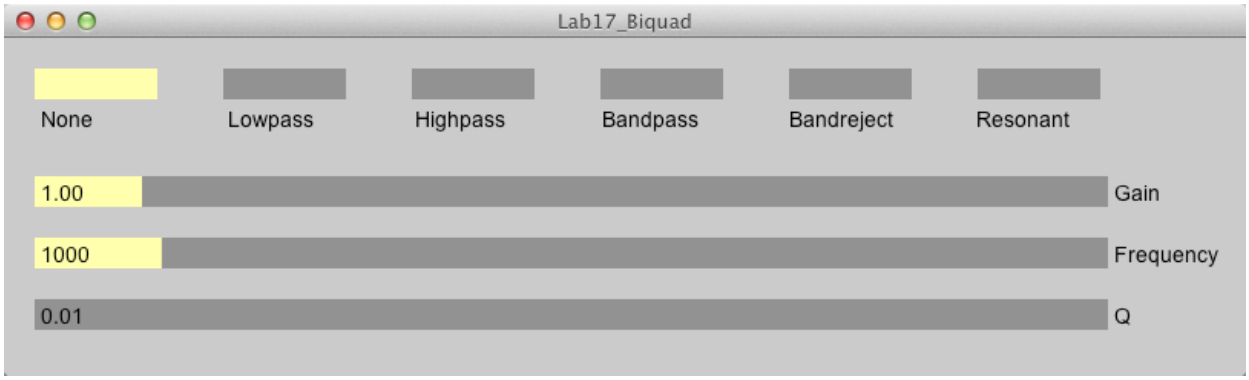
Expressed as a time domain formula, the biquad filter equation is:

$$y[n] = a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] - b_1 y[n-1] - b_2 y[n-2]$$

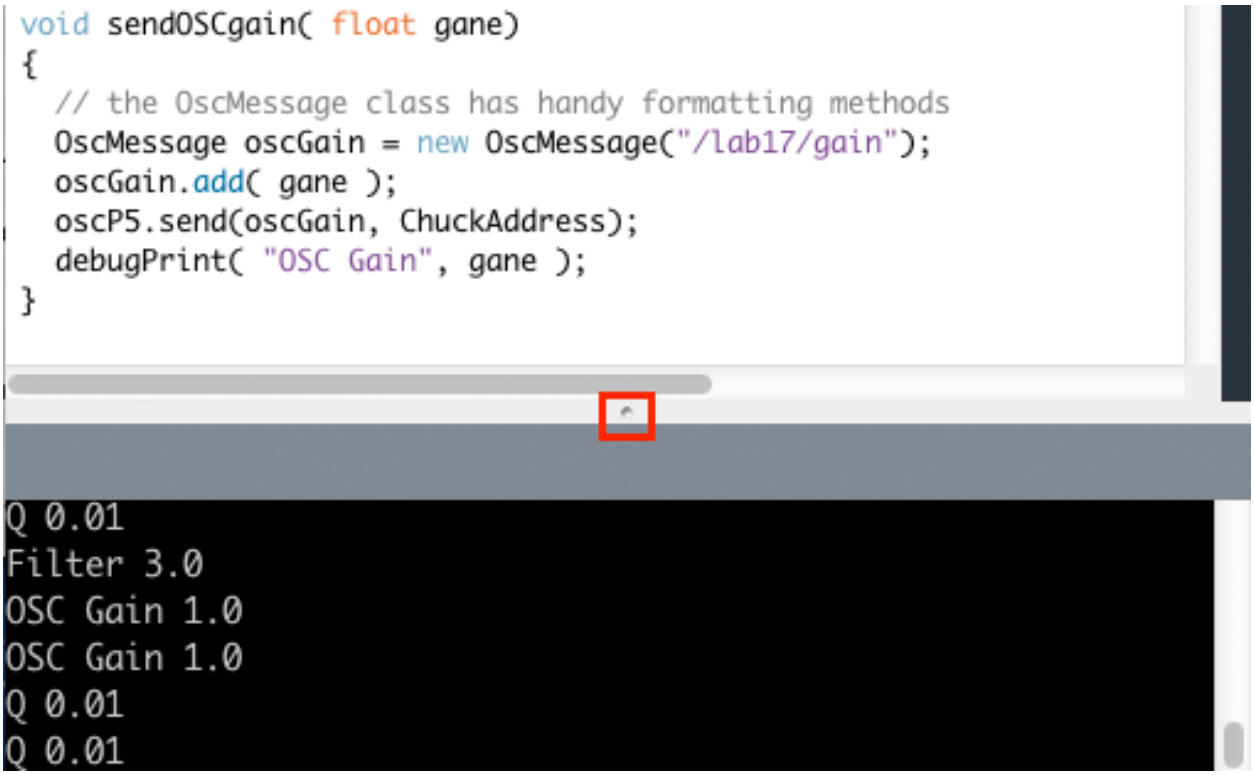
y[n] is the current output sample, x[n] is the current input sample. x[n-1] and x[n-2] are the two previous input samples, and y[n-1] and y[n-2] are the two previous output samples. By choosing appropriate values for a0, a1, a2, b1, and b2 the biquad equation can be used to produce all five common audio filter types; a lowpass, highpass, bandpass, bandreject, or a resonant filter.

Biquad Filter Processing GUI

Open the m208Lab17/Lab17_Biquad folder and run Lab17_Biquad.pde in Processing. Move the sliders and click the filter buttons and watch the output in Processing's Console Monitor. You may have to increase the height of the console window to see the button changes.



The GUI and OpenSoundControl (OSC) commands are fully functional but ChucK is not yet ready to receive them.



Biquad Filter ChucK Implementation

Open a new blank window in miniAudicle. Save it as Lab17_biquad.ck. As you follow along in this Lab17, each block of code should be added to Lab17_biquad.ck as its presented. Enter this code.

```
// Lab17_biquad.ck
// John Ellinger Music 208 Winter 2014

// Debugging Utility
1 => int DEBUG_PRINT;

Impulse imp => dac;
SndBuf buf;
buf.read( me.sourceDir() + "/noise.wav" );
0 => buf.pos;
```

Lab16 used the SndBufUtilsClass to read in the samples. Lab17 uses the SndBuf object directly.

The following variables hold the OSC (OpenSoundControl) message values sent to ChucK as Processing's filter type buttons and Gain, Frequency, and Q sliders are changed. Add this code.

```
// Map Processing variables to ChucK variables
1000.0 => float freq;
1.0 => float gane;
0.01 => float Q;
1.0 => float filterType; // None button is 1
```

We'll use a special type of array in ChucK to map the filter type buttons to self documenting strings. The array index is the name of the button (a string type) and the array value is the button number (an int type). In CS this is known as an Associative Array, a Map, a Dictionary, or Key Value pairs. Here the key is a string and the value is the number.

```
// example of a ChucK associative array indexed by strings
// that can be used to make your code self documenting
int filterRA[6];
1 => filterRA["None"];
2 => filterRA["Lowpass"];
3 => filterRA["Highpass"];
4 => filterRA["Bandpass"];
5 => filterRA["Bandreject"];
6 => filterRA["Resonant"];
```

The Biquad formula uses coefficients a0, a1, a2, b1, and b2, so we'll need variables for them.

$$y[n] = a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] - b_1 y[n-1] - b_2 y[n-2]$$

```
// Biquad coefficients
1.0 => float a0;
0 => float a1;
0 => float a2;
0 => float b1;
0 => float b2;
```

The Biquad formula also needs to remember the two previous inputs, and the two previous outputs so we'll need variables for them.

```
// current and previous inputs and outputs
0 => float nowOutput;
0 => float lastInput;
0 => float lastInput2;
0 => float lastOutput;
0 => float lastOutput2;
```

Create the OscRecv object and port number that listens for OSC messages sent from Processing. The port number must match the one defined in Processing.

```
// create our OSC receiver
OscRecv recv;
// listen on port 12346 that was defined in Processing
12346 => recv.port;
// start listening
recv.listen();
```

Specify which events to listen for. The events must use the exact text message that is sent from Processing.

```
// our four OSC receive events, each receive one float
recv.event( "/lab17/filterType, f" ) @=> OscEvent oeFilterType;
recv.event( "/lab17/gain, f" ) @=> OscEvent oeGain;
recv.event( "/lab17/freq, f" ) @=> OscEvent oeFrequency;
recv.event( "/lab17/q, f" ) @=> OscEvent oeQ;
```

Create the code to process each OSC message. This code determines which filter type button is selected.

```
// The OSC Listening function for filter buttons
function void updateFilterType()
{
    while ( true )
    {
        oeFilterType => now;
        while ( oeFilterType.nextMsg() != 0 )
        {
            oeFilterType.getFloat() => filterType;
            calcCoefficients( filterType );
            if ( DEBUG_PRINT )
                <<< "filterType", filterType >>>;
        }
    }
}
```

Use the code above as a guide and write functions that process the the three slider messages.

```
// The OSC Listening function for sliderGain
function void updateGain()
{
    while ( true )
    {
        // you finish
    }
}
// The OSC Listening function for sliderGain
function void updateFrequency()
{
    while ( true )
    {
        // you finish
    }
}
// The OSC Listening function for sliderGain
function void updateQ()
{
    while ( true )
    {
        // you finish
    }
}
```

The calcCoefficients() function is called whenever the filter buttons, Frequency, or Q sliders change. It must be included in the updateFrequency, and updateQ functions. It is not needed in the updateGain function.

```
function void calcCoefficients( float filterType )
{
    if ( filterType == filterRA["None"] )
        calcNone();
    else if ( filterType == filterRA["Lowpass"] )
        calcLowpass();
    else if ( filterType == filterRA["Highpass"] )
        calcHighpass();
    else if ( filterType == filterRA["Bandpass"] )
        calcBandpass();
    else if ( filterType == filterRA["Bandreject"] )
        calcBandreject();
    else if ( filterType == filterRA["Resonant"] )
        calcResonant();
}
```

All that's left is to determine the coefficients for a0, a1, a2, b1, and b2 for each of the five filter types plus the None button. Add this code to calculate coefficients when the None button is selected.

None or No Filter

$$a_0 = 1.0$$

$$a_1 = a_2 = b_1 = b_2 = 0$$

```
function void calcNone()
{
    if ( DEBUG_PRINT )
        <<< "calcNone" >>>;
    1.0 => a0;
    0 => a1;
    0 => a2;
    0 => b1;
    0 => b2;
}
```

Add stub functions for the other five filter types so we can test the OpenSoundControl messaging between Processing and ChucK. Stub functions simply print a message stating they've been called. We'll add complete the code after we verify that the OSC messages are received correctly.

```
// all formulas from Designing Audio Effect Plug-ins in C++
// http://www.willpirkle.com/about/books/
function void calcLowpass()
{
    if ( DEBUG_PRINT )
        <<< "calcLowpass" >>>;
}

function void calcHighpass()
{
    if ( DEBUG_PRINT )
        <<< "calcHighpass" >>>;
}

function void calcBandpass()
{
    if ( DEBUG_PRINT )
        <<< "calcBandpass" >>>;
}

function void calcBandreject()
{
    if ( DEBUG_PRINT )
        <<< "calcBandreject" >>>;
}
function void calcResonant()
{   if ( DEBUG_PRINT )
        <<< "calcResonant", Q >>>;
}
```

Add code to spork the OSC listeners for each message.

```
// spork'em
spork ~ updateFilterType();
spork ~ updateGain();
spork ~ updateFrequency();
spork ~ updateQ();
```

Write the main loop code that plays the SndBuf. This is the code that will be used once the five filter coefficients are implemented. For now it just plays the original noise.wav file.

```
while ( true )
{
    for ( 0 => int n; n < buf.samples(); n++ )
    {
        a0 *buf.valueAt(n) + a1*lastInput + a2*lastInput2
               - b1*lastOutput - b2*lastOutput2 => nowOutput;

        lastInput => lastInput2;
        lastOutput => lastOutput2;
        buf.valueAt(n) => lastInput;
        nowOutput => lastOutput;

        gane * nowOutput => imp.next;
        1::samp => now;
    }
}
```

Initial Test

Run BiquadFilter.ck in miniAudicle. Fix any errors. When the code is error free run Lab17_Biquad.pde in Processing. Operate all GUI controls. Verify that all GUI control messages are received in miniAudicle's Console Monitor window. The only controls that currently work are Gain slider and the Resonant button. The Resonant button that should reduce the gain, any other button will set it back to normal. The Filter types will work as soon as we implement the code to set the filter coefficients. But before that we need to optimize the code for the non debug version.

Debug Print Functions

Processing's print() and println() functions and ChucK's <<< … >>> print function are extremely useful while debugging your programs. However they take CPU time away from audio processing. In a release version the print statements should be turned off. This is normally done with a variable that turns the messages on or off.

In Lab17_Biquad.pde set dbg to false in the debugPrint() function.

```
void debugPrint( String s, float f )
{
  Boolean dbg = true;
  if ( dbg )
    println( s, f );
}
```

It's used like this.

```
void sendOSCgain( float gane)
{
  // the OscMessage class has handy formatting methods
  OscMessage oscGain = new OscMessage("/lab17/gain");
  oscGain.add( gane );
  oscP5.send(oscGain, ChuckAddress);
  debugPrint( "OSC Gain", gane );
}
```

Set DEBUG_PRINT variable to zero in Lab17_biquad.ck.

```
// Lab17_biquad.ck
// John Ellinger Music 208 Winter 2014

// Debugging Utility
0 => int DEBUG_PRINT;
```

It's used like this.

```
function void calcLowpass()
{
    if ( DEBUG_PRINT )
        <<< "calcLowpass" >>>;
}
```

Biquad Filter Coefficients

Each of the five filter types can be created by changing the values of a0, a1, a2, b1, and b2 in the biquad equation.

$$y[n] = a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] - b_1 y[n-1] - b_2 y[n-2]$$

The following formulas are from the book from Designing Audio Effect Plug-ins in C++. http://www.willpirkle.com/about/books

Lowpass Filter

$$C = \frac{1}{\tan(\pi F / SR)}$$

$$D = C^2 + C\sqrt{2} + 1$$

$$a_0 = \frac{1}{D}$$

$$a_1 = \frac{2}{D}$$

$$a_2 = \frac{1}{D}$$

$$b_1 = \frac{2(1 - C^2)}{D}$$

$$b_2 = \frac{C^2 - C\sqrt{2} + 1}{D}$$

This is the code for the Lowpass filter.

```
// all formulas from Designing Audio Effect Plug-ins in C++
// http://www.willpirkle.com/about/books/
function void calcLowpass()
{
    if ( DEBUG_PRINT )
        <<< "calcLowpass" >>>;
    // Calculate the value for C
    1.0 / Math.tan( ( pi*freq ) / SR ) => float C;
    // Calculate the value for D
    C*C + Math.sqrt(2) * C + 1 => float D;
    // calculate the coefficients
    1.0 / D => a0;
    2.0 / D => a1;
    1.0 / D => a2;
    2.0 * ( 1 - C*C ) / D => b1;
    ( C*C - Math.sqrt(2) * C + 1 ) / D => b2;
}
```

Highpass Filter

$$C = \tan(\pi F / SR)$$

$$D = C^2 + C\sqrt{2} + 1$$

$$a_0 = \frac{1}{D}$$

$$a_1 = \frac{-2}{D}$$

$$a_2 = \frac{1}{D}$$

$$b_1 = \frac{2(C^2 - 1)}{D}$$

$$b_2 = \frac{C^2 - C\sqrt{2} + 1}{D}$$

Implement code for the Highpass filter.

```
function void calcHighpass()
{
    if ( DEBUG_PRINT )
        <<< "calcHighpass" >>>;
}
```

Bandpass Filter

$$C = \tan(\pi F / SR)$$

$$D = C^2 Q + C + Q$$

$$a_0 = \frac{C}{D}$$

$$a_1 = 0$$

$$a_2 = \frac{-C}{D}$$

$$b_1 = \frac{2Q(C^2 - 1)}{D}$$

$$b_2 = \frac{C^2 Q - C + Q}{D}$$

Implement code for the Bandpass filter.

```
function void calcBandpass()
{
    if ( DEBUG_PRINT )
        <<< "calcBandpass" >>>;
}
```

Bandreject Filter

$$C = \tan(\pi F / SR)$$

$$D = C^2 Q + C + Q$$

$$a_0 = \frac{Q(1 + C^2)}{D}$$

$$a_1 = \frac{2Q(C^2 - 1)}{D}$$

$$a_2 = a_0$$

$$b_1 = a_1$$

$$b_2 = \frac{C^2 Q - C + Q}{D}$$

Implement code for the Bandreject filter.

```
function void calcBandreject()
{
    if ( DEBUG_PRINT )
        <<< "calcBandreject" >>>;
}
```

Resonant Filter

$$C = 2Q\cos(2\pi F / SR)$$

$$a_0 = 0.5 - \frac{Q^2}{2}$$

$$a_1 = 0$$

$$a_2 = C$$

$$b_1 = -C$$

$$b_2 = Q^2$$

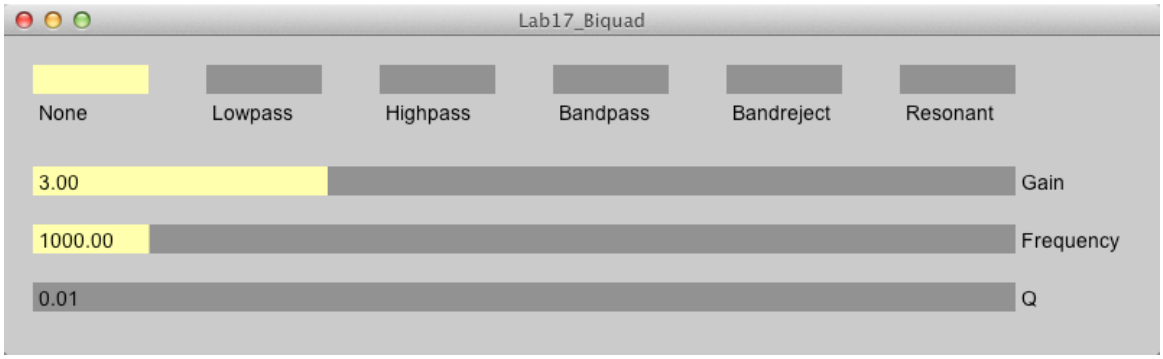
Implement code for the Resonant filter.

```
function void calcResonant()
{
    // Processing scales Gain to 0.01-0.5
    // Processing scales Q to 0.9-0.9999
    if ( DEBUG_PRINT )
        <<< "calcResonant", Q >>>;
}
```

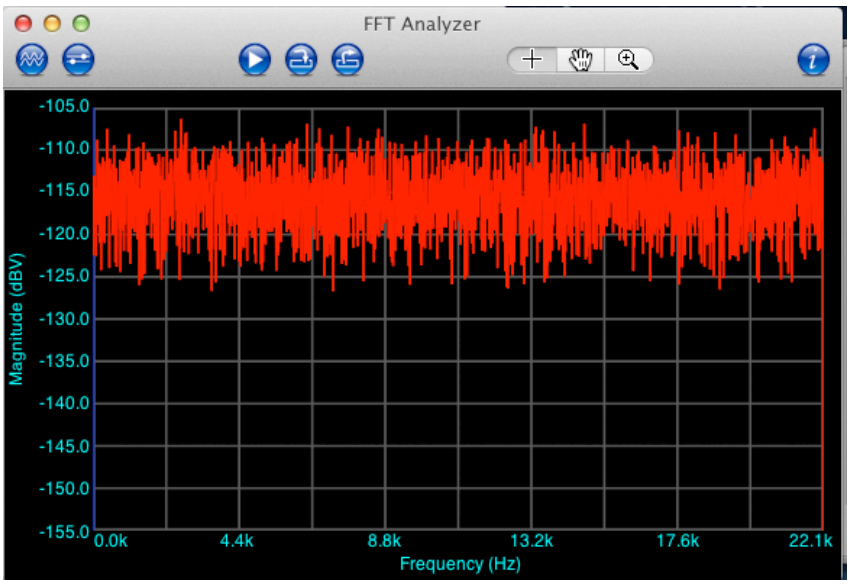
Crunch Time

TAKE OFF YOUR HEADPHONES AND TURN DOWN THE VOLUME.

Run the program.



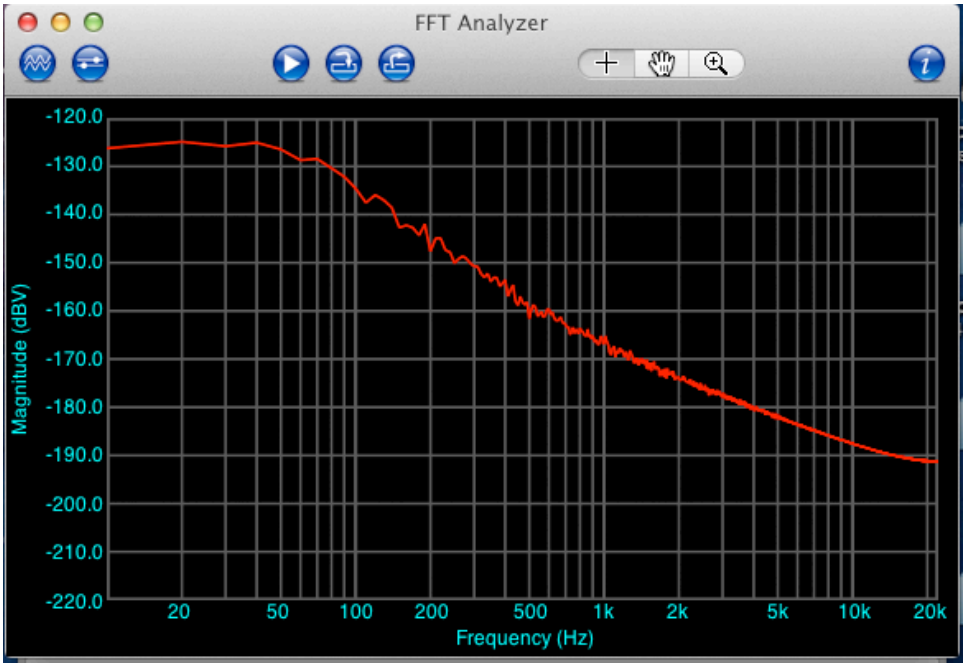
You should see this in SoundScope.



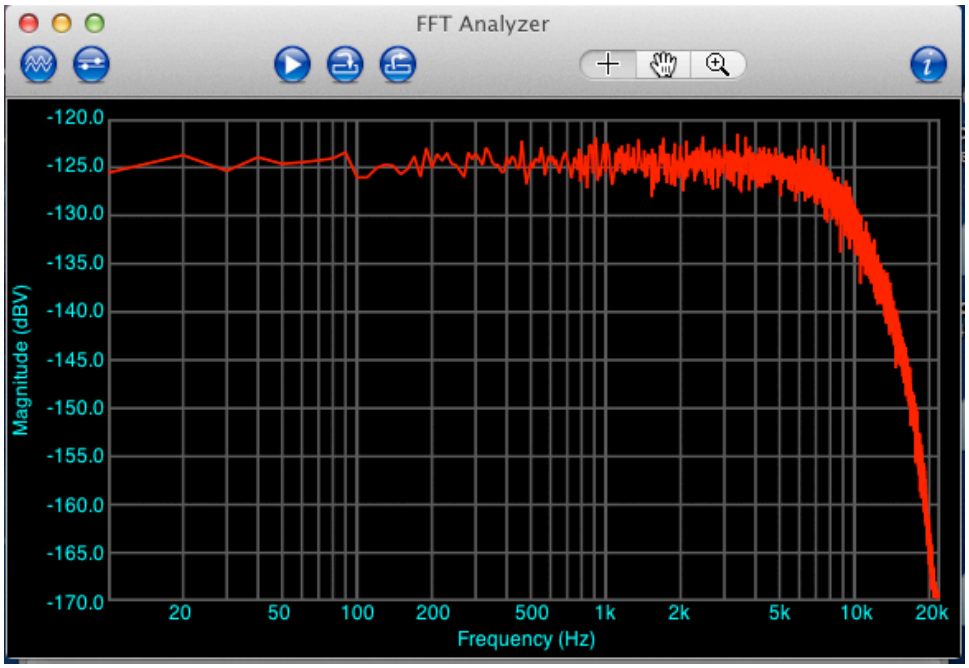
Lowpass Test

The Gain and Frequency sliders are functional. Sweep the frequency control.

Frequency at 60 Hz



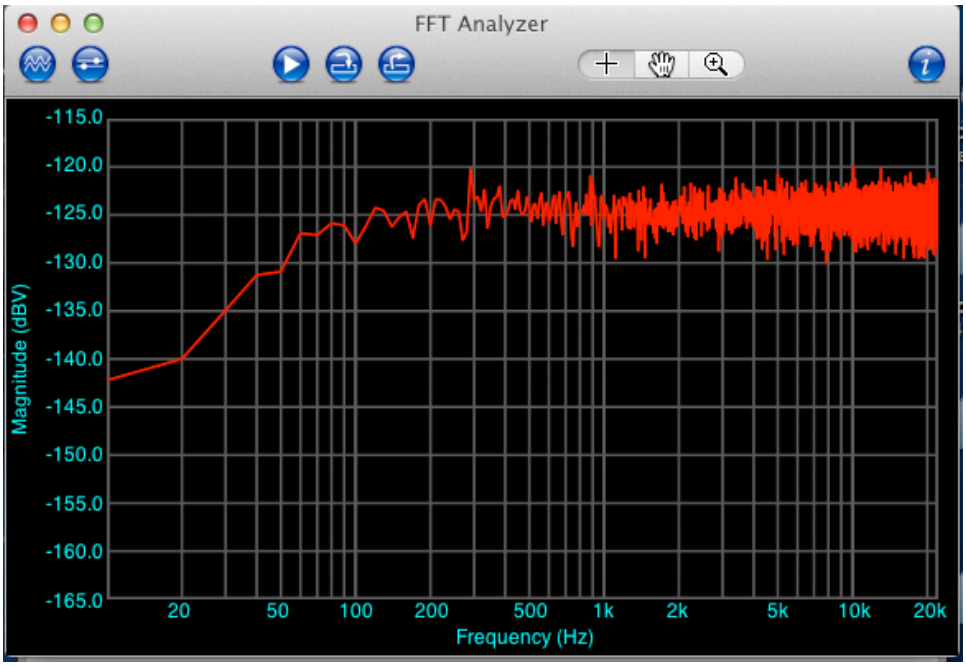
Frequency at 8000 Hz



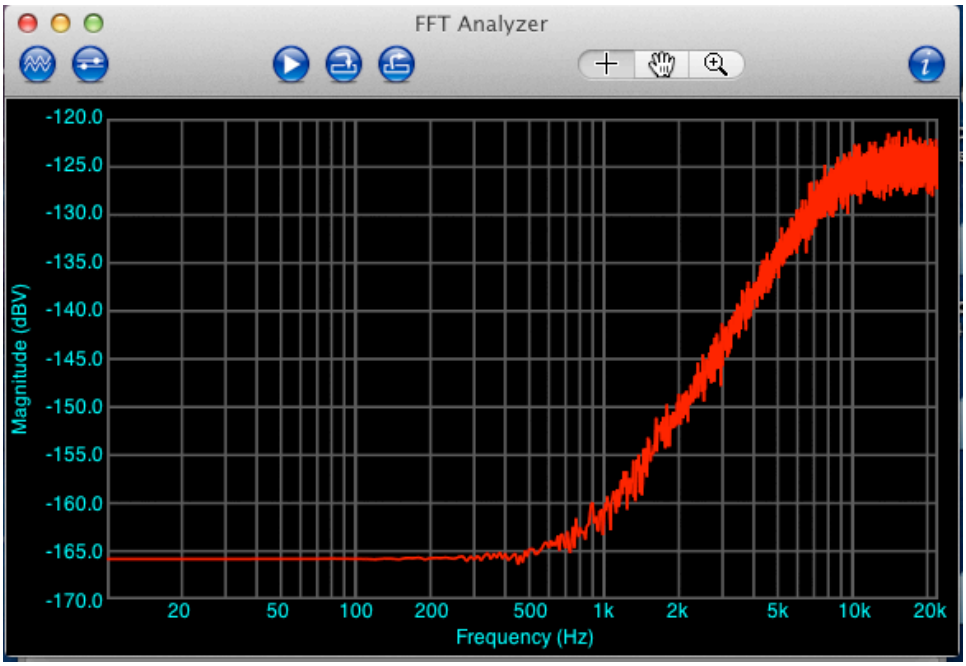
Highpass Test

The Gain and Frequency sliders are functional. Sweep the frequency control.

Frequency at 60 Hz



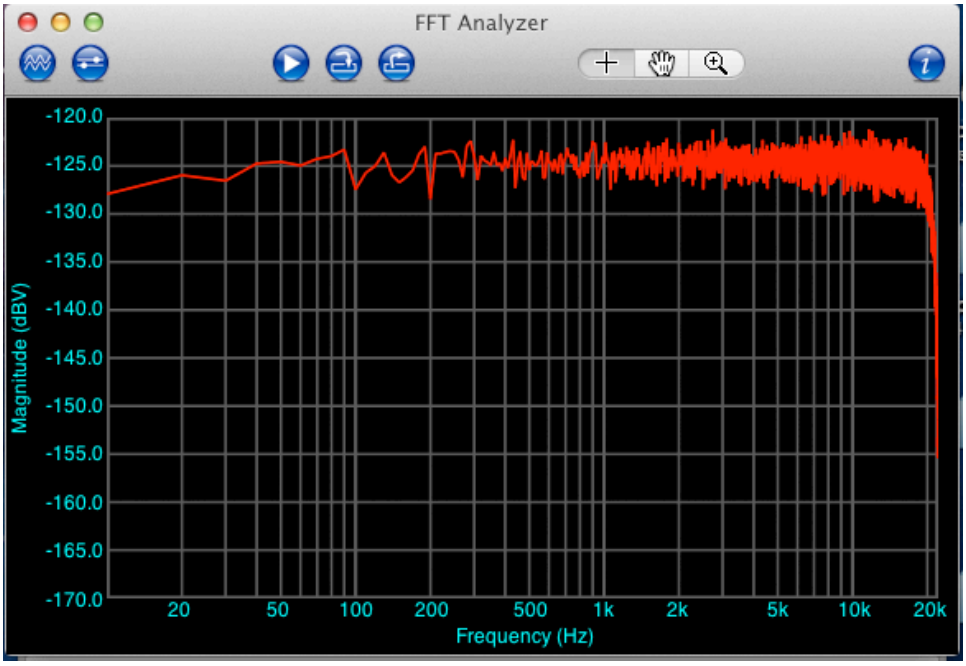
Frequency at 8000 Hz



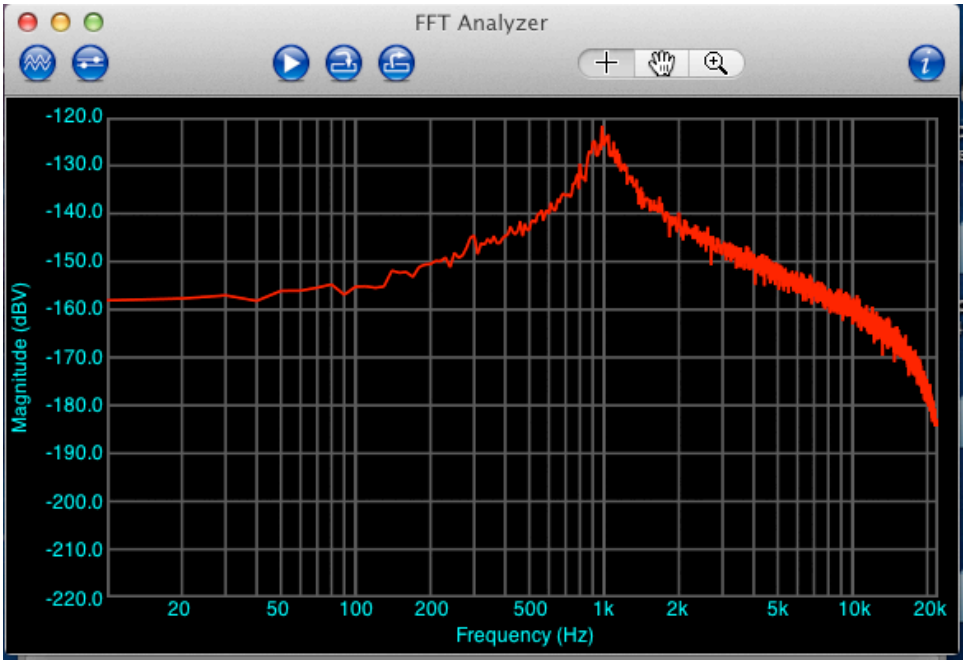
Bandpass Test

All three sliders are functional. Set the Frequency to 1000. Sweep the Q control

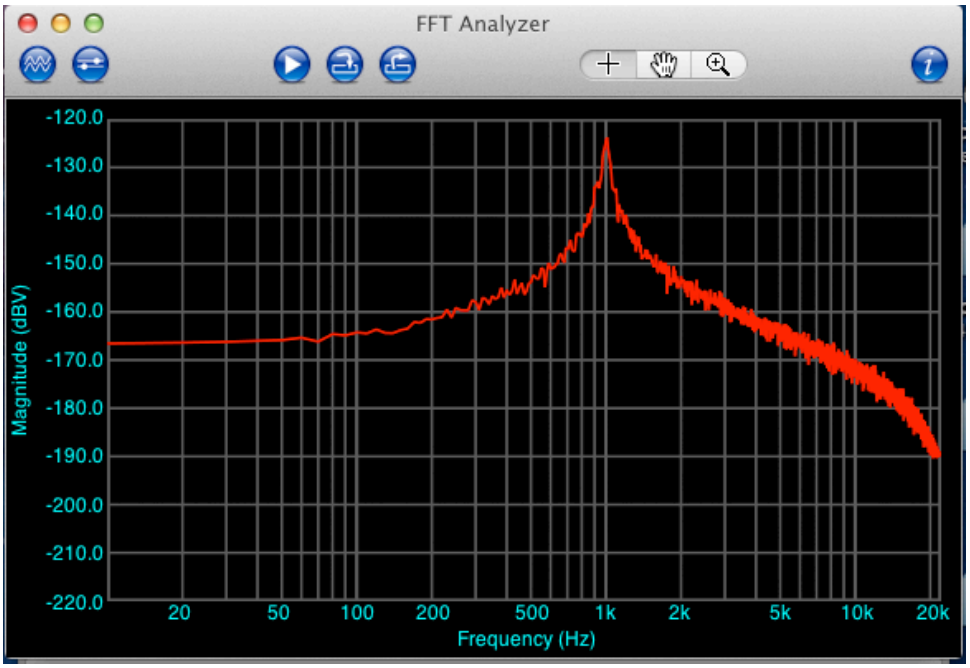
Q at 0.01



Q around 5.0

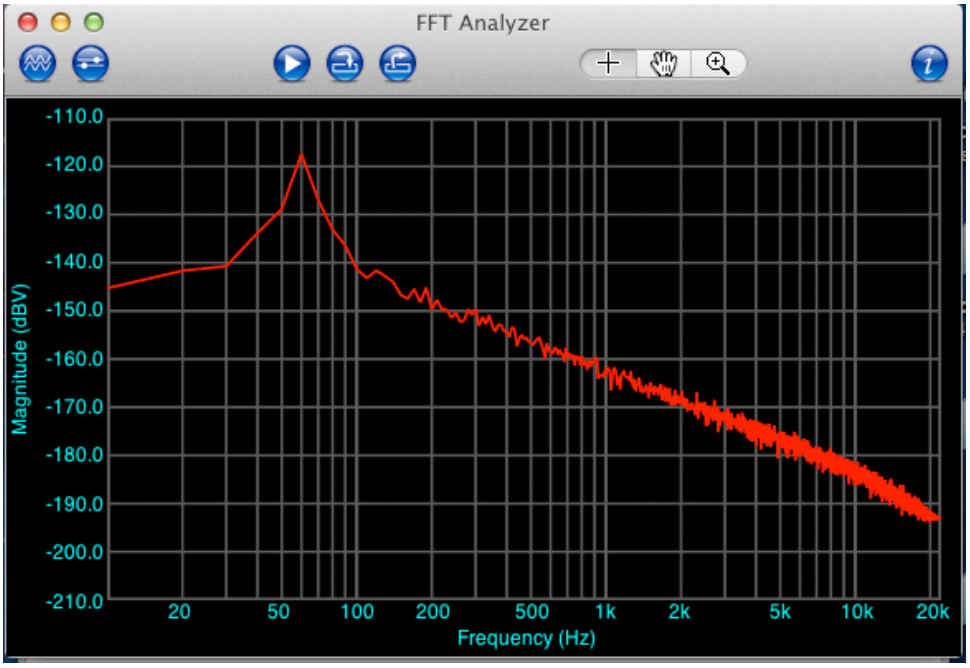


Q at 20

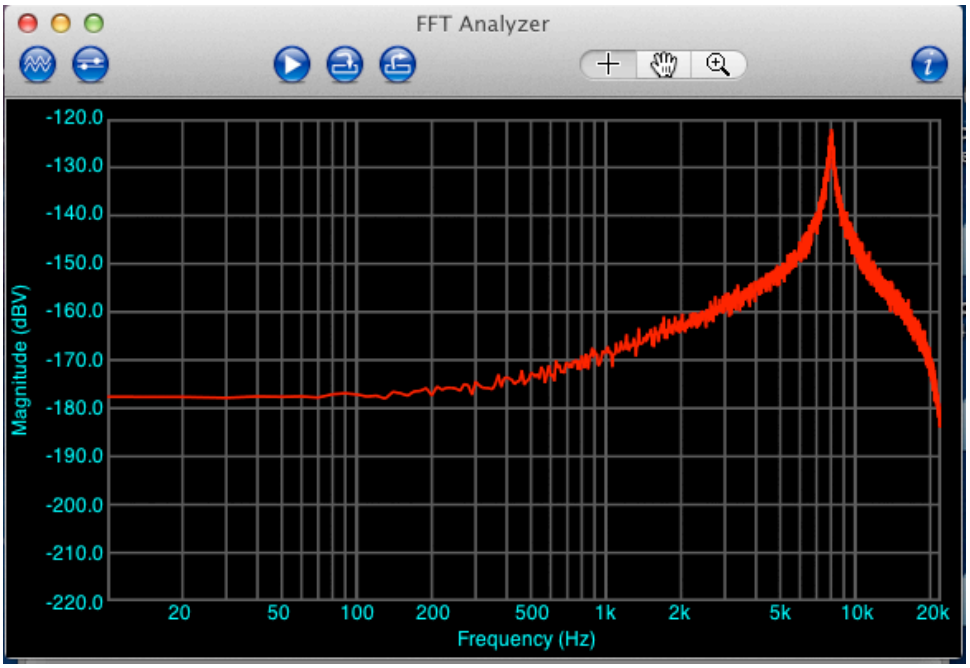


Leave the Q at 20 and sweep the frequency

Frequency at 60 Hz



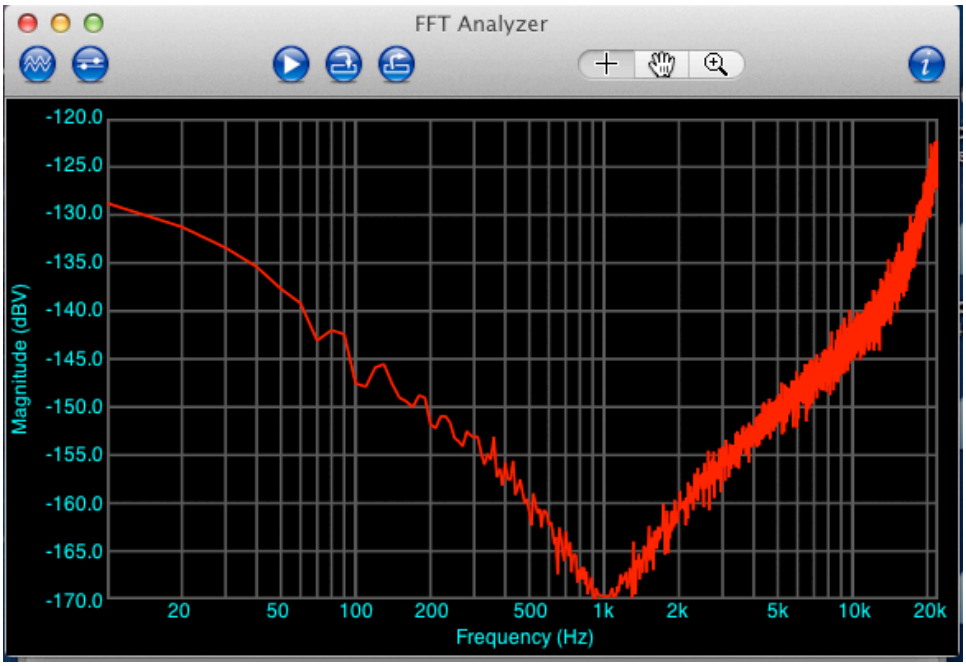
Frequency at 8000 Hz



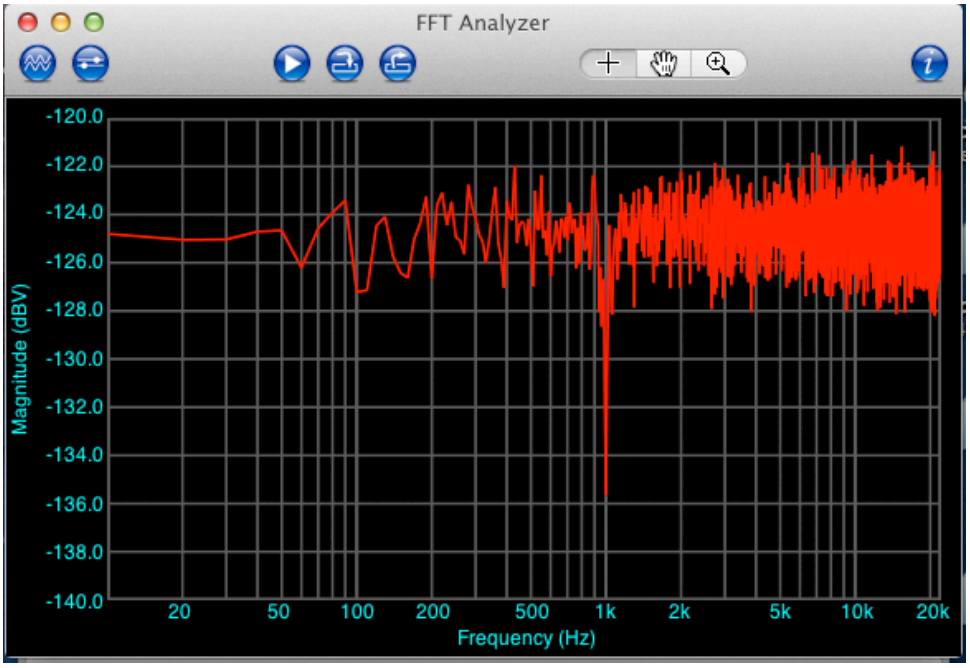
Bandreject Test

All three sliders are functional. Set the Frequecy to 1000. Sweep the Q control. It will work opposite to the Bandpass. Lower Q values show the greatest rejection.

Q at 0.01

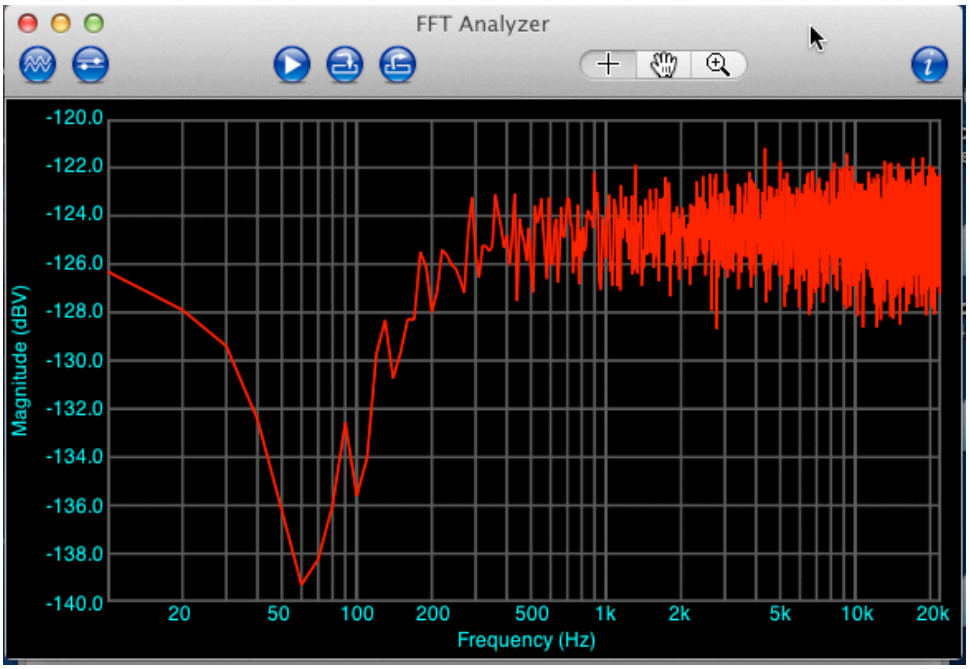


Q at 20

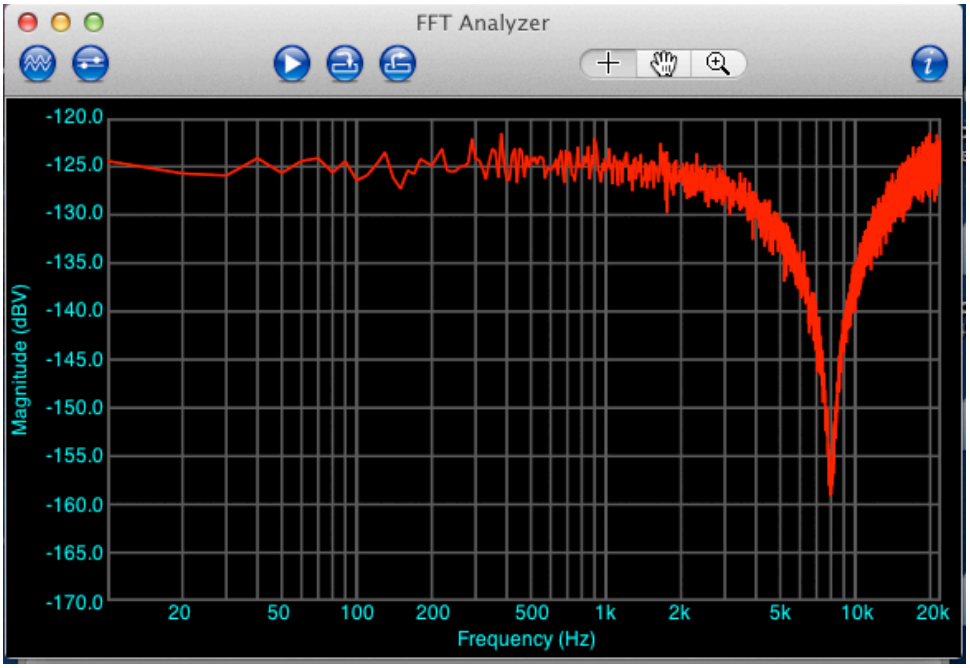


Set the Q at 0.40 and sweep the frequency.

Frequency at 60 Hz



Frequency at 8000 Hz



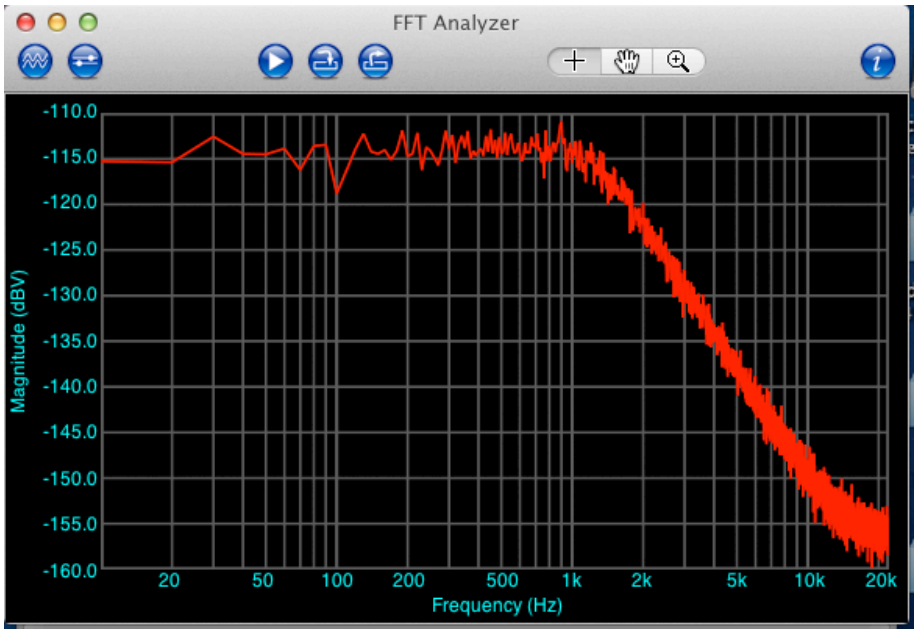
Resonant Test

WARNING: This filter is unstable and can provide extremely loud feedback very quickly. I've tried to mitigate the feedback by altering the sliderQ range (0.9 - 0.9999) and reducing the amplitude to 0.05 when the Resonant button is clicked.

```
void controlEvent(ControlEvent theEvent)
{
    if (theEvent.isFrom( filterBtns ) )
    {
      filterType = theEvent.getValue();
      sendOSCFilterType( filterType );
      if ( filterType == 6 ) // Resonant
      {
        sliderGain.setRange(0.01, 0.5).setDecimalPrecision(2).setValue(0.051);
        sliderQ.setRange(0.9, 0.9999).setDecimalPrecision(4).setValue(0.9);
      }
      else
      {
        sliderGain.setRange(0, 10).setDecimalPrecision(2).setValue(1.0);
        sliderQ.setRange(0.01, 20).setDecimalPrecision(2).setValue(0.01);
      }
    }
```

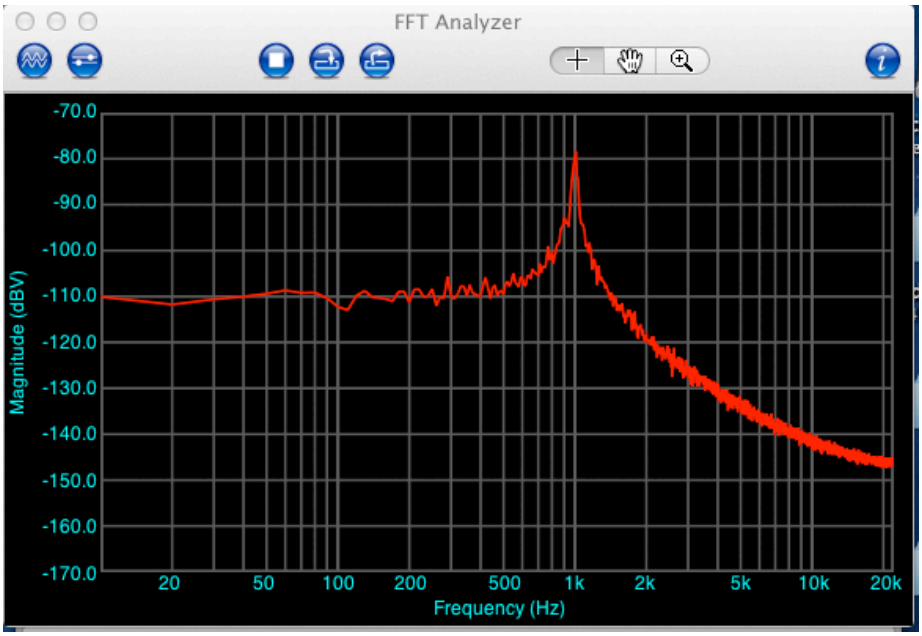
All three sliders are functional. Set the frequency to 1000 Hz and sweep the Q control. When Q is 0.9 or below the spectrum resembles a lowpass filter.

Q at 0.9



Above 0.96 you can start to see and hear resonance.

Q at 0.9985



When you've found a tolerable resonance sweep the frequency.

Stars And Stripes March

Test the filters on a short musical example: the piccolo solo from John Phillip Sousa's march The Stars and Strips Forever.

```
347 Impulse imp => dac;
348 SndBuf buf;
349 //buf.read( "Noise.wav" );
350 buf.read( "StarsNStripes.wav" );
351 0 => buf.pos;
```

Click the None button to hear the original wav file. Notice the piccolo solo frequency range is centered around 2300 Hz.

Lowpass Filter

At frequencies below 500 Hz the piccolo is can hardly be heard.

Highpass Filter

At frequencies above 2000 the piccolo is prominent. Around 4500 Hz you start hearing a piccolo and snare drum duet.

Bandpass Filter

Set the frequency around 2300 Hz and sweep the Q. You can do a pretty good job of soloing the piccolo.

Bandreject Filter

Set the frequency around 2300 Hz and set the Q to 0.01. You can almost eliminate the piccolo.

Resonant Filter

Be careful with this one. Keep the Gain VERY LOW. Set the frequency around 2300 and move the Q almost to the right edge above 0.99+. You should hear the piccolo take on a distinctly metallic tone as the filter resonates.