MUSC 208 Winter 2014
John Ellinger Carleton College

# Lab 16 Filters

## Lab 16 needs to be done on the iMacs.

The only reason we need to use the iMacs is because of the Signal Scope application that will be used towards the end of the lab to display a real time FFT of the audio that's being played. All code created in Processing and ChucK is cross platform and will work on both Mac and Windows. I'm certain there is an equivalent Windows application for the Mac only SignalScope but you'll have to discover that on your own.

## Setup

Download the m208Lab16.zip files.

## Overview

Finite Impulse Response (FIR) Filters

Moving Average Filter

Low Pass and High Pass filters

Processing GUI to Experiment with

An audio filter can be thought of as a black box that has an input and an output. Digital samples enter through the input. Some type of processing occurs on a single sample or on a group of samples. The processed samples are then sent to the output ready to be played through a speaker. The output may have a strong resemblance the input or have not resemblance at all.

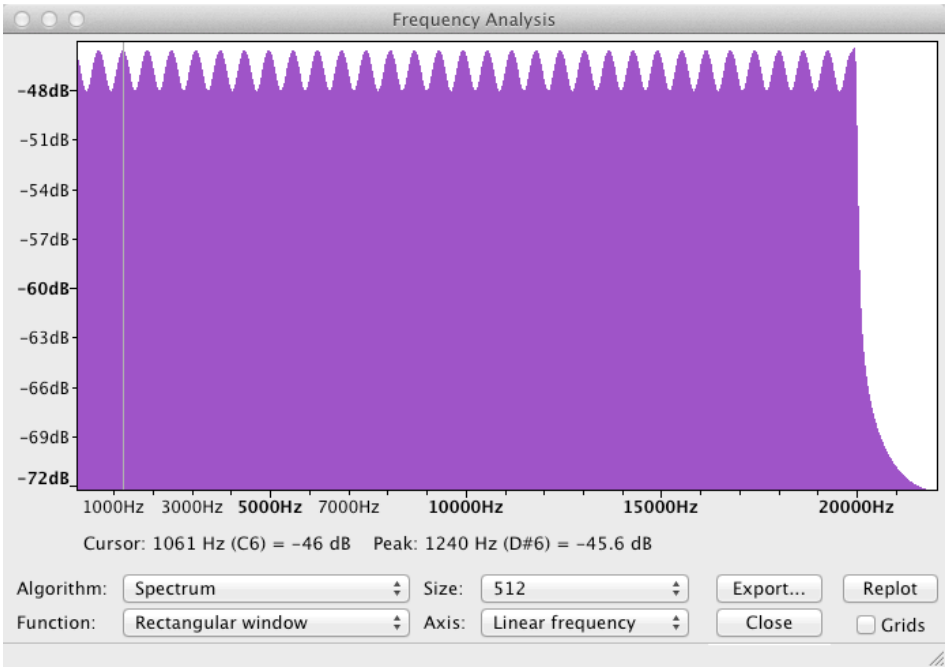
Lab 16 will look at the two basic digital filter types, the Finite Impulse Response filter and the Infinite Impulse Response filter. The FIR filters depend only on current and previous samples while IIR filters also depend on past output. IIR filters can become unstable if the output is fed back into the input and causes an exponential increase in volume similar to holding a live microphone close to an

output speaker.

Create a wave file with a fundamental frequency of 100 Hz and 200 harmonics all with equal amplitude that covers the frequency spectrum from 100 Hz - 20,000 Hz. When we apply a filter to this waveform it will be easy to see the results. Enter and execute this code.

```
// 200sines.ck
// John Ellinger, Music 208, Spring2013

200 => int NUM_HARMONICS;
SinOsc sinRA[ NUM_HARMONICS ]; // array of 200 Sine oscillators
Gain g => dac => WvOut w => blackhole; // no audio output
me.sourceDir() + "/sines20K.wav" => w.wavFilename; // filename to save

100 => float f0; // fundamental frequency
1.0 / NUM_HARMONICS => float amp; // scale amplitude
for ( 0 => int n; n < sinRA.cap(); n++ )
{
    f0 * ( n +1 )=> sinRA[n].freq;
    amp => sinRA[n].gain;
     // send output to Gain object that will mix all SinOsc's together
     sinRA[n] => g;
}
1::second => now; // waveform will be one second long
```

When the program exits, open the sines20k.wav file in Audacity and look at the frequency spectrum.




## Finite Impulse Response Filter

FIR filters only depend on current and previous samples. The general form of an FIR filter is:

$$output[n] = a_0 sample[n] + a_1 sample[n-1] + ..a_M sample[n-M]$$

*or*

$$output[n] = \sum_{i=0}^{M} a_i sample[n-i]$$

## Moving Average Filter

We'll look at a very simple FIR filter, called the moving average filter. The output of the filter is the average of the current and previous sample.

### Low Pass

$$output[n] = 0.5 \bullet sample[n] + 0.5 \bullet sample[n-1]$$

### High Pass

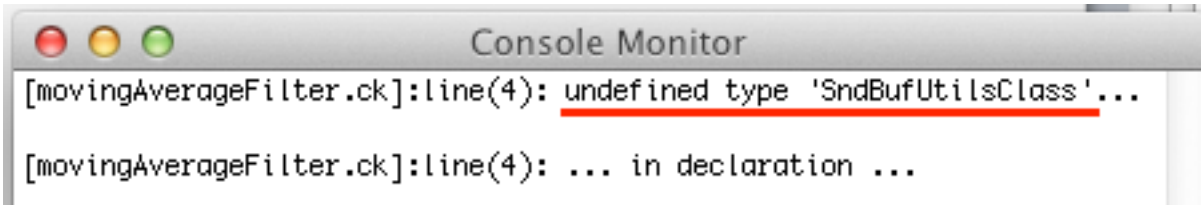
$$output[n] = 0.5 \bullet sample[n] - 0.5 \bullet sample[n-1]$$

Create this code.

```
// movingAverageFilter.ck
// John Ellinger Music 208 Winter2014

SndBufUtilsClass sbc;
Impulse imp => dac => WvOut w => blackhole;

sbc.init( me.sourceDir() + "/sines20k.wav" );
 me.sourceDir() + "/lowPassMovingAverage.wav" => w.wavFilename;
0.5 => float a0;
0.5 => float a1;
```

```
for ( 0 => int n; n < sbc.lenSamples; n++ )
{
    if ( n == 0 )
        a0 * sbc.sampRA[0] => imp.next;
    else // low pass uses + between a0 and a1
        (a0 * sbc.sampRA[n] ) + (a1 * sbc.sampRA[n-1] ) =>
sbc.sampRA[n];

    sbc.sampRA[n] => imp.next;
    1::samp => now;
}

sbc.init(  me.sourceDir() + "/sines20k.wav" );
 me.sourceDir() + "/highPassMovingAverage.wav" => w.wavFilename;
0.5 => a0;
0.5 => a1;
for ( 0 => int n; n < sbc.lenSamples; n++ )
{
    if ( n == 0 )
        a0 * sbc.sampRA[0] => imp.next;
    else  // high pass uses - between a0 and a1
        (a0 * sbc.sampRA[n] ) - (a1 * sbc.sampRA[n-1] ) =>
sbc.sampRA[n];

    sbc.sampRA[n] => imp.next;
    1::samp => now;
}
null @=> w;
```

Run it and you'll get an error.




There's an updated version of SndBufUtilsClass.ck in the m208Lab16 folder. There are three ways to run the movingAverageFilter.ck code without causing the error. Try all three.

## Method 1. miniAudicle

Stop and restart the miniAudicle Virtual Machine. Run the SndBufUtilsClass.ck first and the movingAverageFilter.ck code second.

## Method 2. Terminal

Open Terminal. Set the working directory to the m208Lab16 folder and execute these commands.

```
509,je@jemac:~/Desktop/m208Lab16$ cd ~/Desktop/m208Lab16
510,je@jemac:~/Desktop/m208Lab16$ ls
SndBufUtilsClass.ck    movingAverageFilter.ck sines20K.wav
511,je@jemac:~/Desktop/m208Lab16$ chuck SndBufUtilsClass.ck movingAverageFilter.ck
512,je@jemac:~/Desktop/m208Lab16$
```

## Method 3. Use the Machine.add() function

Create a new ChucK file called runMovingAverage.ck. Enter and run this code.

```
// runMovingAverage.ck
// John Ellinger Music 208 Winter2014

Machine.add( me.sourceDir() + "/SndBufUtilsClass.ck" );
Machine.add( me.sourceDir() + "/movingAverageFilter.ck" );
```

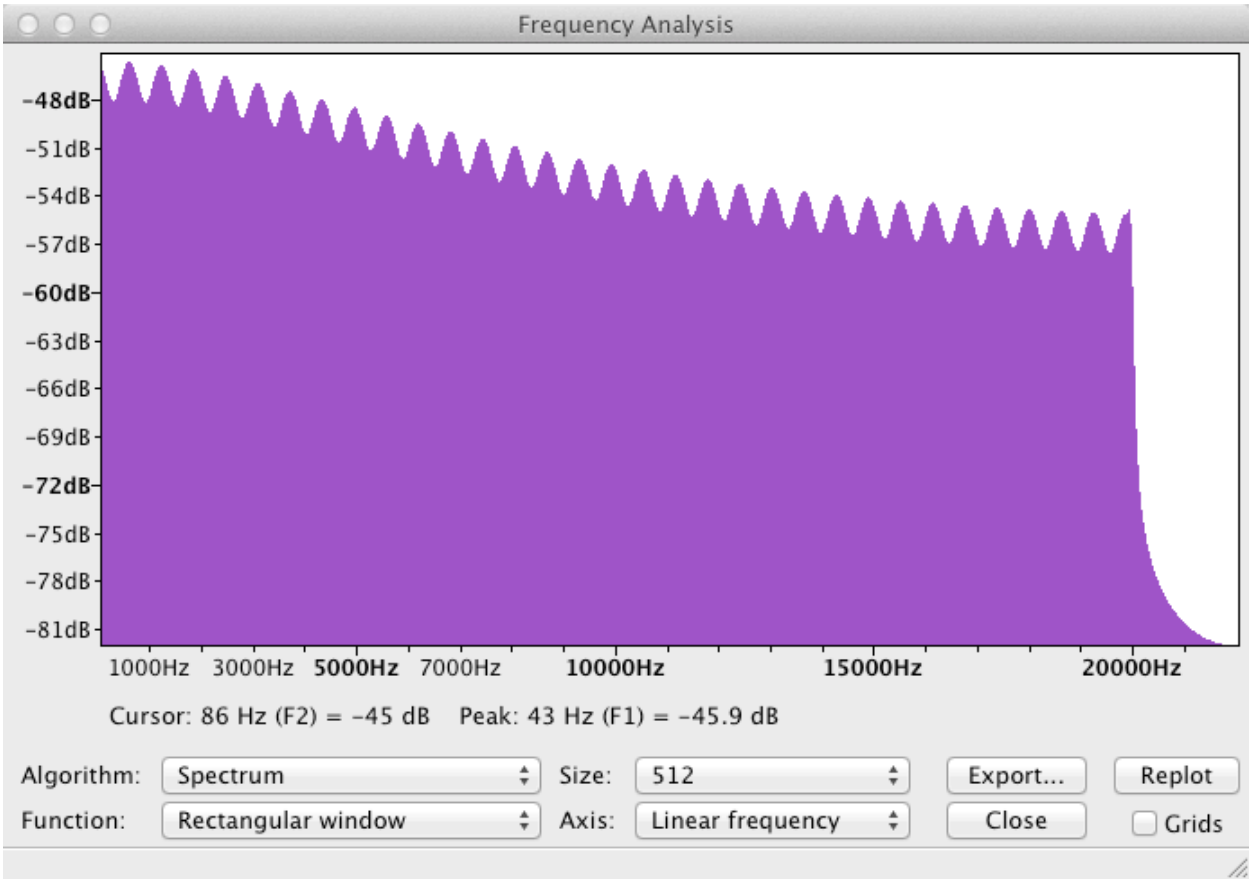
Two new wave files should be in the m208Lab16 folder.

```
512,je@jemac:~/Desktop/m208Lab16$ ls
SndBufUtilsClass.ck        lowPassMovingAverage.wav  sines20K.wav
highPassMovingAverage.wav  movingAverageFilter.ck
```

Open the two wave files in Audacity and look at their spectrums.

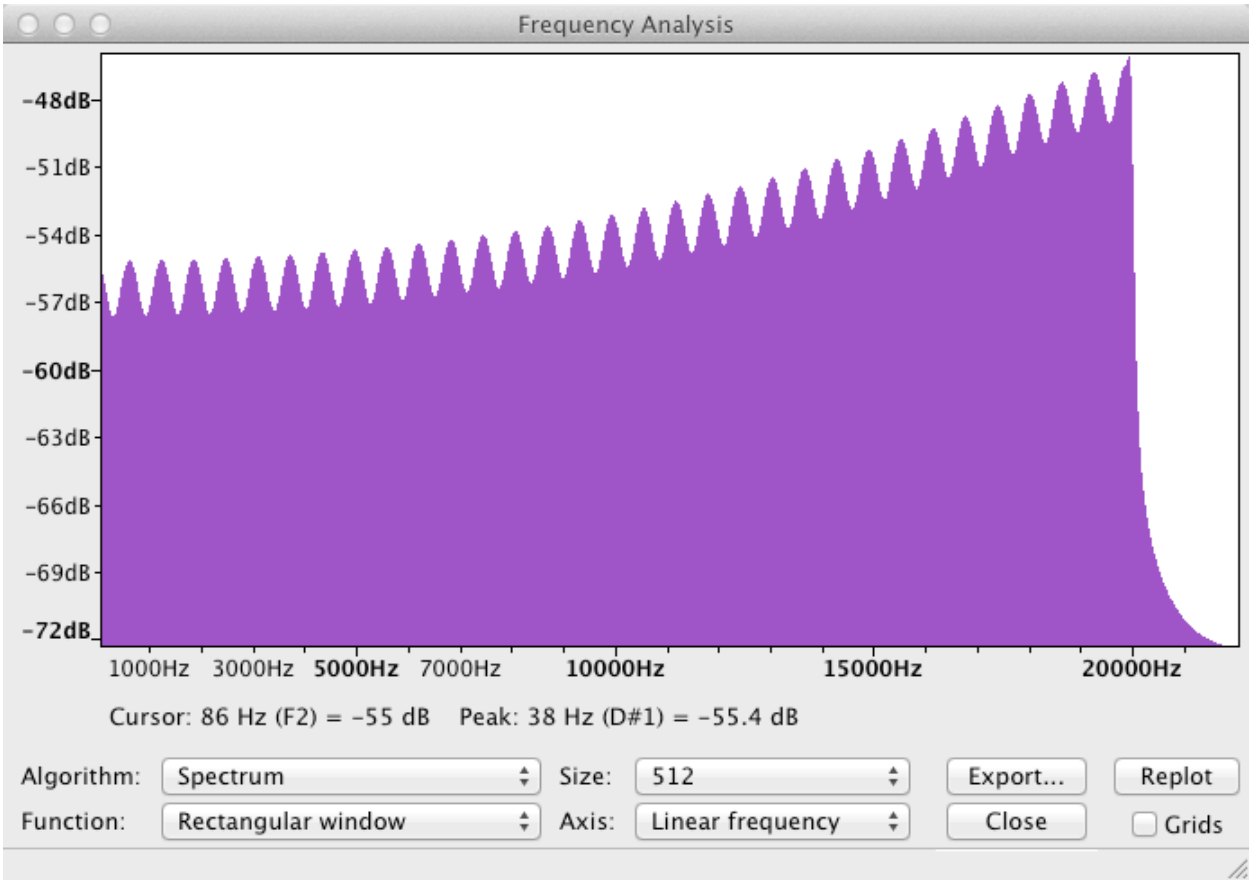
## Low Pass Spectrum

The low pass filter allows low frequencies to pass through but attenuates high frequencies.




## High Pass Spectrum

The high pass filter allows high frequencies to pass through but attenuates low frequencies.




## More Flexible Low Pass And High Pass Filters

If you vary the coefficients of the moving average filter, the frequency response will change. If you add the samples you'll get a low pass filter. If you subtract the samples you'll get a high pass filter.

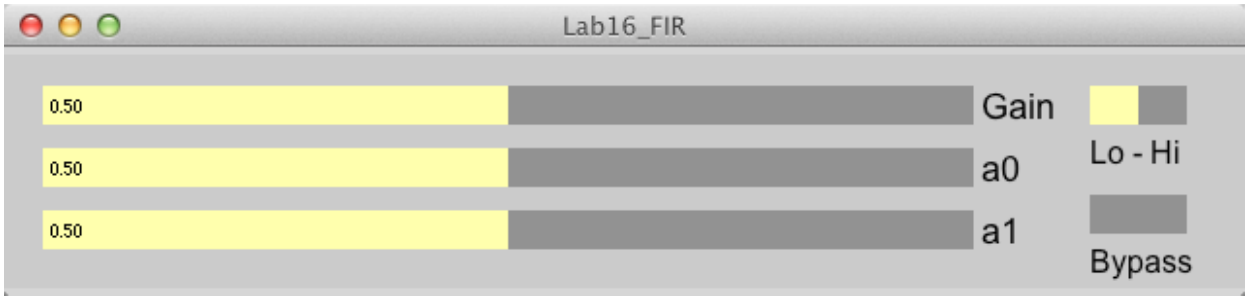
$$output[n] = a_0 \bullet sample[n] \ \pm \ a_1 \bullet sample[n-1]$$

where $a_0$ and $a_1$ sum to 1.0 to avoid clipping

## Processing => ChucK => SoundScope

We'll investigate this filter by building a GUI in Processing to manipulate the coefficients, send their values to ChucK where hear the filter effects, and also send the audio to the SoundScope FFT Analyzer application so we can see the results.

When completed, the Processing GUI you'll build will look like this.




Open the Processing Application and save your sketch as "Lab16_FIR" in the m208Lab16 folder.

## Step 1 Create the Gain Slider

Enter and run this code in Processing.

Imports and global variables.

```
// Lab16_FIR.pde
// MUSC208 John Ellinger Winter2014

import oscP5.*;
import netP5.*;
import controlP5.*;

ControlP5 cp5;
// the controlP5 controls
Slider sliderGain;
```

Next define the Gain control settings.

```
// GUI control layout
void addGUIControls()
{
  // create the font used for for control labels
  // use true/false for anti-aliased/not anti-aliased
  PFont pfont = createFont("Arial", 18, true);
  ControlFont font = new ControlFont (pfont );

  // slider dimensions
  int sliderHt = 20;    // slider height
  int sliderWid = 480;  // slider width
  // control default colors
  int bkgColor = color(128);            // light gray
  int foreColor = color(255,255,162);   // light yellow
  int activeColor = color(255,255,162); // light yellow

    // create slider Gain
  sliderGain = cp5.addSlider( "Gain" )
    .setPosition(20, 16)              // 20 pixels in from left, 16 down from top
    .setSize( sliderWid, sliderHt )   // 480 pixels wide by 20 pixels tall
    .setRange( 0, 1 )                 // slider minimum value is 0, max is 1.0
    .setValue( 0.5 )                  // initial setting at 0.5
    .setColorCaptionLabel(0)          // caption color is black
    .setColorValueLabel(0)            // label color is black
    .setDecimalPrecision(2)           // display 2 decimal places
    .setColorBackground( bkgColor )   // dark gray
    .setColorForeground( foreColor )  // light yellow
    .setColorActive( activeColor )    // light yellos
    ;
  // change control label font and size
  cp5.getController("Gain")           // operate on the Gain control
     .getCaptionLabel()               // get the control label
     .setFont(font)                   // set the font to Arial 18
     .toUpperCase(false)              // all upper case is false
     .setSize(18)                     // font size is 18
     ;
} // end addGUIControls
```

Create the setup() function. The setup function is the first function called by the Processing application. The size() function must be the first line in setup().s

```
void setup()
{
   // set size of window
   // MUST BE first line in setup
  size( 640, 120 );           // window will be 640 pixels wide by 18 pixels tall
  noStroke();                 // no aliasing
  cp5 = new ControlP5(this);  // create the cp5 object
  addGUIControls();           // add the controls
}
```

Create the draw() function. Processing calls the draw() function whenever the screen needs updating.

```
void draw()
{
  // width and height are Processing variables passed to
  // the the size() function in in setup()
  rect(0, 0, width, height);
  background( color(192) );  // a gray window background
}
```

Create the controlEvent() function that handles user interaction with the controlP5 controls you've created.

```
void controlEvent(ControlEvent theEvent)
{
  if ( theEvent.isFrom( sliderGain ) )
  {
    float gane = sliderGain.getValue();
    println( "Gain", gane );
  }
}
```

Run the program. You should see this.

Move the slider and watch the output in the Processing console.

```
Gain 0.56875
Gain 0.5708334
Gain 0.57708335
Gain 0.5791667
Gain 0.5833334
```

## Step2 Create the OSC event that will be sent to ChucK

Add these lines of code to the globals section.

```
ControlP5 cp5;
// the controlP5 controls
Slider sliderGain;

// OscP5 is name of class, oscP5 is a variable instance of that class
OscP5 oscP5;
// ChucK network address
// on the same computer it will be "127.0.0.1"
// or "localhost"
NetAddress ChuckAddress;

// GUI control layout
void addGUIControls()
```

Add the OSC setup commands in the setup() function.

```
void setup()
{
   // set size of window
   // MUST BE first line in setup
  size( 640, 120 );           // window will be 640 pixels wide by 18 pixels tall
  noStroke();                 // no aliasing
  cp5 = new ControlP5(this);  // create the cp5 object
  addGUIControls();           // add the controls
  // OSC stuff
  // if ChucK were to send back to Processing it would send to port 12345
  oscP5 = new OscP5(this,  12345);
  // Processing will send to ChucK on port 12346
  // 127.0.0.1 is the network address of the local machine
  ChuckAddress = new NetAddress("127.0.0.1",  12346);
}
```

Create the function that will send the OSC message

```
void draw()
{
  // width and height are Processing variables passed to
  // the the size() function in in setup()
  rect(0, 0, width, height);
  background( color(192) );  // a gray window background
}

void sendOSCgain( float gane)
{
  // the OscMessage class has handy formatting methods
  // genereate the message title ChucK will use to identify
  // the control that sent the message
  OscMessage oscGain = new OscMessage("/lab16/gain");
  // add one float value to the message
  // the parameter gane is the current value of the slider
  oscGain.add( gane );
  //send the message immediately
  oscP5.send(oscGain, ChuckAddress);
}

void controlEvent(ControlEvent theEvent)
{
```

Modify the controlEvent so the sendOSCgain() function is called when the slider is moved.

```
void controlEvent(ControlEvent theEvent)
{
  if ( theEvent.isFrom( sliderGain ) )
  {
    float gane = sliderGain.getValue();
    sendOSCgain( gane );
    println( "OSC sent gain", gane );
  }
}
```

## Step 3 Create the OSC receive messages in ChucK

Create a new ChucK file named simpleFIR.ck. Enter this code.

**Important: Turn down the computer volume. The noise.wav is loud.**

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

```
Impulse imp; // define but don't connect yet
SndBufUtilsClass sbc;
sbc.init( me.sourceDir() + "/noise.wav" );
// copy the SndBuf samples from the SndBufUtilsClass
// into a local array of samples called ra.
sbc.getSamples() @=> float ra[];
ra.cap() => int numSamps;
// sanity check
<<< "numSamps", numSamps >>>;

// Set the gain to the initial Processing
0.5 => float gane;

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

// listen for the /lab16/gain event followed by one float
recv.event( "/lab16/gain, f" ) @=> OscEvent oeGain;

// The OSC Listening function for sliderGain
function void updateGain()
{
    while ( true )
    {
        // wait for event to arrive
        oeGain => now;
        // grab the next message from the queue
        // it will be one float - the sliderGain value
        while ( oeGain.nextMsg() != 0 )
        {
            oeGain.getFloat() => gane;
            <<< "sliderGain", gane >>>;
        }
    }
}

// spork the OSC listener
spork ~ updateGain();
```

```
// connect Impulse to dac
imp => dac;
// our counter
0 => int n;
while (true)
{
    if ( n == 0 )
        gane * ra[0] => imp.next;
    else
        gane * ra[n] => imp.next; // low pass

    1::samp => now;
    n++;
    if ( n == numSamps )
        0 => n;
}
```

Create a new file in ChucK that will run both SndBufUtilsClass and the simpleFIR.

```
// runSimpleFIR.ck

Machine.add( me.sourceDir() + "/SndBufUtilsClass.ck" );
Machine.add( me.sourceDir() + "/simpleFIR.ck" );
```

## Step 4 Implement the A0 slider in Processing.

Add the sliderA0 variable to the global variables list. You might as well add the sliderA1 variable at the same time.

```
ControlP5 cp5;
// the controlP5 controls
Slider sliderGain;
Slider sliderA0;
Slider sliderA1;
```

Duplicate (copy/paste) the sliderGain section of of addGUIControls(). Make a few changes.

```
// create slider A0
sliderA0 = cp5.addSlider( "a0" ) // a0 is the control label
   .setPosition(20, XX)             // XX means you figure it out
   .setSize( sliderWid, sliderHt )   // 480 pixels wide by 20 pixels tall
   .setRange( 0, 1 )                 // slider minimum value is 0, max is 1.0
   .setValue( 0.5 )                  // initial setting at 0.5
   .setColorCaptionLabel(0)          // caption color is black
   .setColorValueLabel(0)            // label color is black
   .setDecimalPrecision(2)           // display 2 decimal places
   .setColorBackground( bkgColor )   // dark gray
   .setColorForeground( foreColor )  // light yellow
   .setColorActive( activeColor )    // light yellos
   ;
// change control label font and size
cp5.getController("a0")           // operate on the Gain control
    .getCaptionLabel()               // get the control label
    .setFont(font)                   // set the font to Arial 18
    .toUpperCase(false)              // all upper case is false
    .setSize(18)                     // font size is 18
    ;
```

## Step 5 Implement the A1 slider in Processing

Do the same thing to create sliderA1.

## Step 6 Create the OSC message function for the two sliders

```
// the sendOSCa0 function will be used for both sliderA0 and sliderA1
// they're linked so that they always sum to 1.0
void sendOSCa0( float a0 )
{
  //the OscMessage class has handy formatting methods
  OscMessage oscA0 = new OscMessage("/lab16/a0"); //greate the address
  oscA0.add( a0 );//append these items
  oscA0.add( 1 - a0 );//append these items
  oscP5.send(oscA0, ChuckAddress);//send the message
}
```

## Step 7 Handle the controlEvent when the sliders are moved

```
void controlEvent(ControlEvent theEvent)
{
  if ( theEvent.isFrom( sliderGain ) )
  {
    float gane = sliderGain.getValue();
    sendOSCgain( gane );
    println( "OSC sent gain", gane );
  }
  else if ( theEvent.isFrom( sliderA0 ) )
  {
    float a0 = sliderA0.getValue();
    sliderA1.setValue( 1-a0 );
    sendOSCa0( a0 );
    println( "a0_a0a1", a0, 1-a0 );
  }
  else if ( theEvent.isFrom( sliderA1 ) )
  {
    float a1 = sliderA1.getValue();
    sliderA0.setValue( 1-a1 );
    float a0 = sliderA0.getValue();
    sendOSCa0( a0 );
    println( "a1_a0a1", a0, 1-a0 );
  }
}
```

## Step 8 Create the ChucK code to process the OSC events

Create variables for the a0 and a1 filter coefficients and add a new receive event.

```
// Set the gain to the initial Processing
0.5 => float gane;
0.5 => float a0;
0.5 => float a1;

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

// listen for the /lab16/gain event followed by one float
recv.event( "/lab16/gain, f" ) @=> OscEvent oeGain;
// listen for the /lab16/a0 event followed by two floats
recv.event( "/lab16/a0, f f" ) @=> OscEvent oeA0;
```

Add the OSC listening function for the the a0 event and spork it.

```
function void updateSliders()
{
    while ( true )
    {
        // wait for event to arrive
        oeA0 => now;
        // grab the next message from the queue.
        // it will be two floats - the values of a0 and a1
        while ( oeA0.nextMsg() != 0 )
        {
            oeA0.getFloat() => a0;
            oeA0.getFloat() => a1;
            <<< "a0", a0, "\ta1", a1  >>>;
        }
    }
}

// spork the OSC listener
spork ~ updateGain();
spork ~ updateSliders();

// connect Impulse to dac
imp => dac;
```

Write the code for the simpleFIR filter.

```
// connect Impulse to dac
imp => dac;

// our counter
0 => int n;
while (true)
{
    if ( n == 0 )
        gane * ra[0] => imp.next;
    else
        gane * ( a0 * ra[n] + a1 * ra[n-1] ) => imp.next; // low pass

    1::samp => now;
    n++;
    if ( n == numSamps )
        0 => n;
}
```

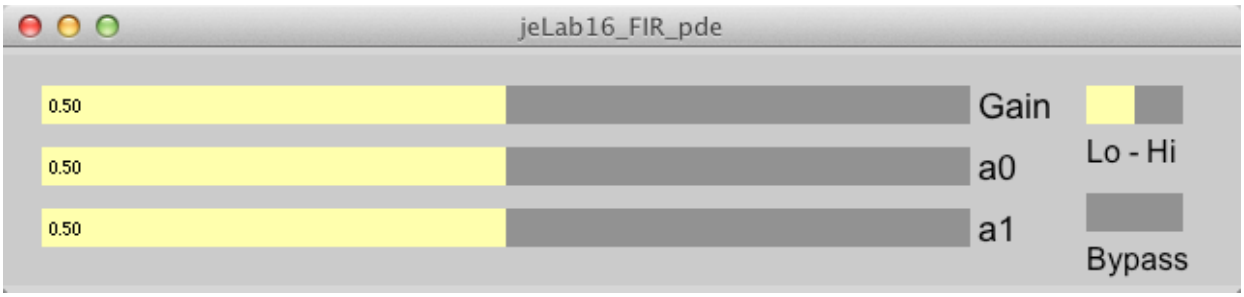
Run the Processing Controls GUI and then runSimpleFIR.ck. Move the sliders and listen for changes in the sound. I did not hear much difference with the low pass filter but it might just be aging ears. I definitely heard a difference with the high pass filter that we'll do next. Stop ChucK.

Change one line to create the high pass filter and run the programs again.

```
    if ( n == 0 )
        gane * ra[0] => imp.next;
    else
        gane * ( a0 * ra[n] - a1 * ra[n-1] ) => imp.next; // high pass
```

## Step 9 GUI Enhancemets

We'll add a two Toggle controls to our Processing GUI, one to switch between low pass and high pass filters and one to bypass the filter and play the original samples.




Add these variables to the global controls.

```
ControlP5 cp5;
// the controlP5 controls
Slider sliderGain;
Slider sliderA0;
Slider sliderA1;
Toggle lohiToggle;
Toggle bypassToggle;
```

Add this code to the addGUIControls function. First the Lo-Hi switch.

```
// create a toggle
 lohiToggle =cp5.addToggle("Lo - Hi")
    .setPosition(XX,YY) // XX YY means you figure out placement in window
    .setSize(XX,YY)     // XX YY means you figure out width height in window
    .setValue(true)
    .setMode(ControlP5.SWITCH) // left/right switch mode
    .setColorBackground(color(128))
    .setColorForeground(color(255,255,162))
    .setColorActive(color(255,255,162))
  ;
  cp5.getController("Lo - Hi")
    .getCaptionLabel()
    .setFont(font)
    .toUpperCase(false)
    .setSize(XX)  // XX means you figure out font size
    .setColor(0)
    ;
```

Then the Bypass button.

```
 // create a toggle
  bypassToggle =cp5.addToggle("Bypass")
     .setPosition(XX,YY) // XX YY means you figure out placement in window
     .setSize(XX,YY)     // XX YY means you figure out width height in window
     .setValue(false)
     .setColorBackground(color(128))
     .setColorForeground(color(128))
     .setColorActive(color(255,255,162))
   ;
   cp5.getController("Bypass")
     .getCaptionLabel()
     .setFont(font)
     .toUpperCase(false)
     .setSize(XX)  // XX means you figure out font size
     .setColor(0)
     ;
} // end addGUIControls
```

Add the OSC message functions.

```
void sendOSCloHi( float lo )
{
  //the OscMessage class has handy formatting methods
  OscMessage oscLoHi = new OscMessage("/lab16/lowhi"); //greate the address
  if ( lo == 1.0 )
    oscLoHi.add( 1.0 ); // plus for low pass
   else
    oscLoHi.add( -1.0 ); // minus for high pass
  oscP5.send(oscLoHi, ChuckAddress);//send the message
}

void sendOSCbypass( float byp )
{
  //the OscMessage class has handy formatting methods
  OscMessage oscBypass = new OscMessage("/lab16/bypass"); //greate the address
  if ( byp == 1.0 )
    oscBypass.add( 1.0 );//append these items
  else
    oscBypass.add( 0.0 );//append these items

   oscP5.send(oscBypass, ChuckAddress);//send the message
}
```

Add the control event handling code.

```
void controlEvent(ControlEvent theEvent)
{
  if ( theEvent.isFrom( sliderGain ) )
  {
    float gane = sliderGain.getValue();
    sendOSCgain( gane );
    println( "OSC sent gain", gane );
  }
  else if ( theEvent.isFrom( sliderA0 ) )
  {
    float a0 = sliderA0.getValue();
    sliderA1.setValue( 1-a0 );
    sendOSCa0( a0 );
    println( "a0_a0a1", a0, 1-a0 );
  }
  else if ( theEvent.isFrom( sliderA1 ) )
  {
    float a1 = sliderA1.getValue();
    sliderA0.setValue( 1-a1 );
    float a0 = sliderA0.getValue();
    sendOSCa0( a0 );
    println( "a1_a0a1", a0, 1-a0 );
  }
  else if ( theEvent.isFrom( lohiToggle ) )
  {
    float lo = lohiToggle.getValue();
    sendOSCloHi( lo );
    println( "lohi ", lo );
  }
  else if ( theEvent.isFrom( bypassToggle ) )
  {
    float byp = bypassToggle.getValue();
    sendOSCbypass( byp );
    println( "bypass ", byp );
  }
}
```

Test the controls in Processing.

## Step 10 Create the ChucK code to process the OSC events

Create variables for the bypass and low high buttons.

```
// Set the gain to the initial Processing
0.5 => float gane;
0.5 => float a0;
0.5 => float a1;
1.0 => float lowhi;
0.0 => float bypass;
```

Add a new receive events.

```
// listen for the /lab16/gain event followed by one float
recv.event( "/lab16/gain, f" ) @=> OscEvent oeGain;
// listen for the /lab16/a0 event followed by two floats
recv.event( "/lab16/a0, f f" ) @=> OscEvent oeA0;
recv.event( "/lab16/lowhi, f" ) @=> OscEvent oeLowHi;
recv.event( "/lab16/bypass, f" ) @=> OscEvent oeBypass;
```

Add the OSC listening function for the the low high event and spork it.

```
function void updateLowPassHiPass()
{
    while ( true )
    {
        // wait for event to arrive
        oeLowHi => now;
        // grab the next message from the queue.
        while ( oeLowHi.nextMsg() != 0 )
        {
            oeLowHi.getFloat() => lowhi;
            if ( lowhi == 1.0 )
                <<< "low pass", lowhi  >>>;
            else if ( lowhi == -1.0 )
                <<< "high pass", lowhi  >>>;
            else
                <<< "bad pass" >>>;
        }
    }
}

// spork the OSC listener
spork ~ updateGain();
spork ~ updateSliders();
spork ~ updateLowPassHiPass();
```

Add the OSC listening function for the the bypass event and spork it.

```
function void updateBypass()
{
    while ( true )
    {
        // wait for event to arrive
        oeBypass => now;
        // grab the next message from the queue.
        while ( oeBypass.nextMsg() != 0 )
        {
            oeBypass.getFloat() => bypass;
            <<< "bypass", bypass  >>>;
        }
    }
}

// spork the OSC listener
spork ~ updateGain();
spork ~ updateSliders();
spork ~ updateLowPassHiPass();
spork ~ updateBypass();
```

Update the code for the simpleFIR filter.

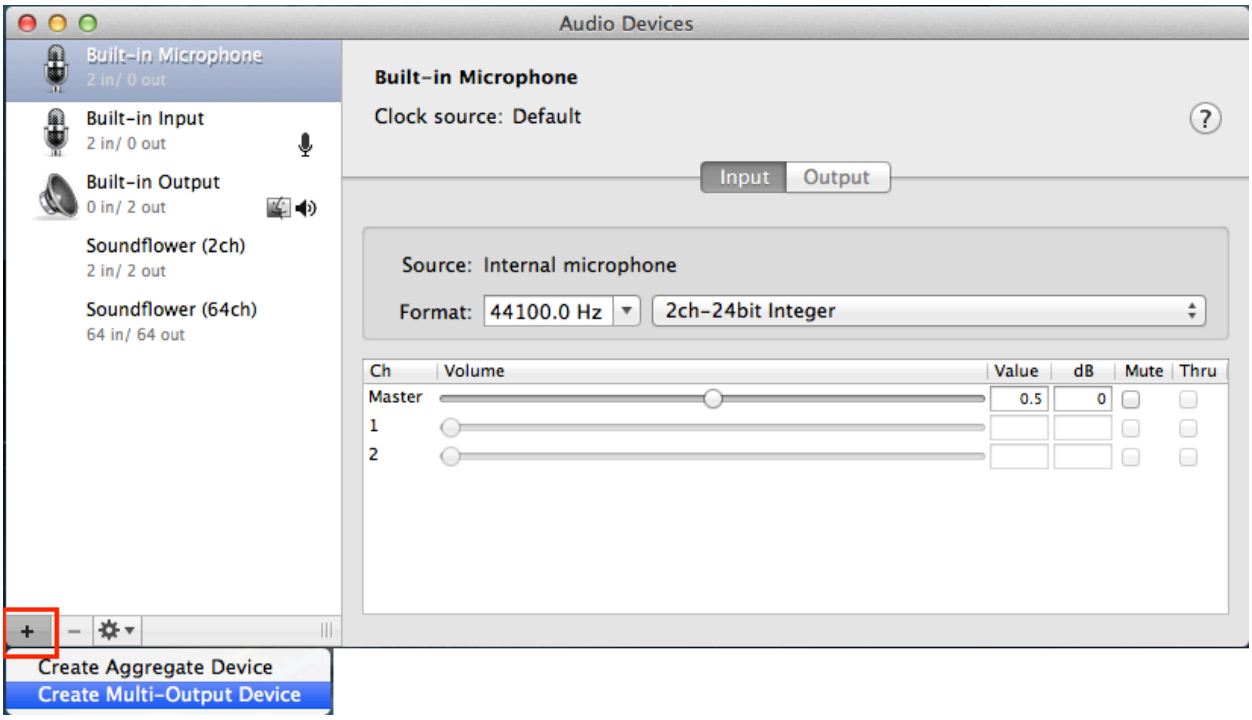
```
// connect Impulse to dac
imp => dac;

// our counter
0 => int n;
while (true)
{
    if ( n == 0 )
        gane * ra[0] => imp.next;
    else
        gane * ( (a0 * ra[n] ) + lowhi * (a1 * ra[n-1] ) ) => imp.next;

    if ( bypass )
        gane * ra[n] => imp.next;

    1::samp => now;
    n++;
    if ( n == numSamps )
        0 => n;
}
```

Run the Processing Controls GUI and then runSimpleFIR.ck. Test the Low Pass High Pass switch and the Bypass button.

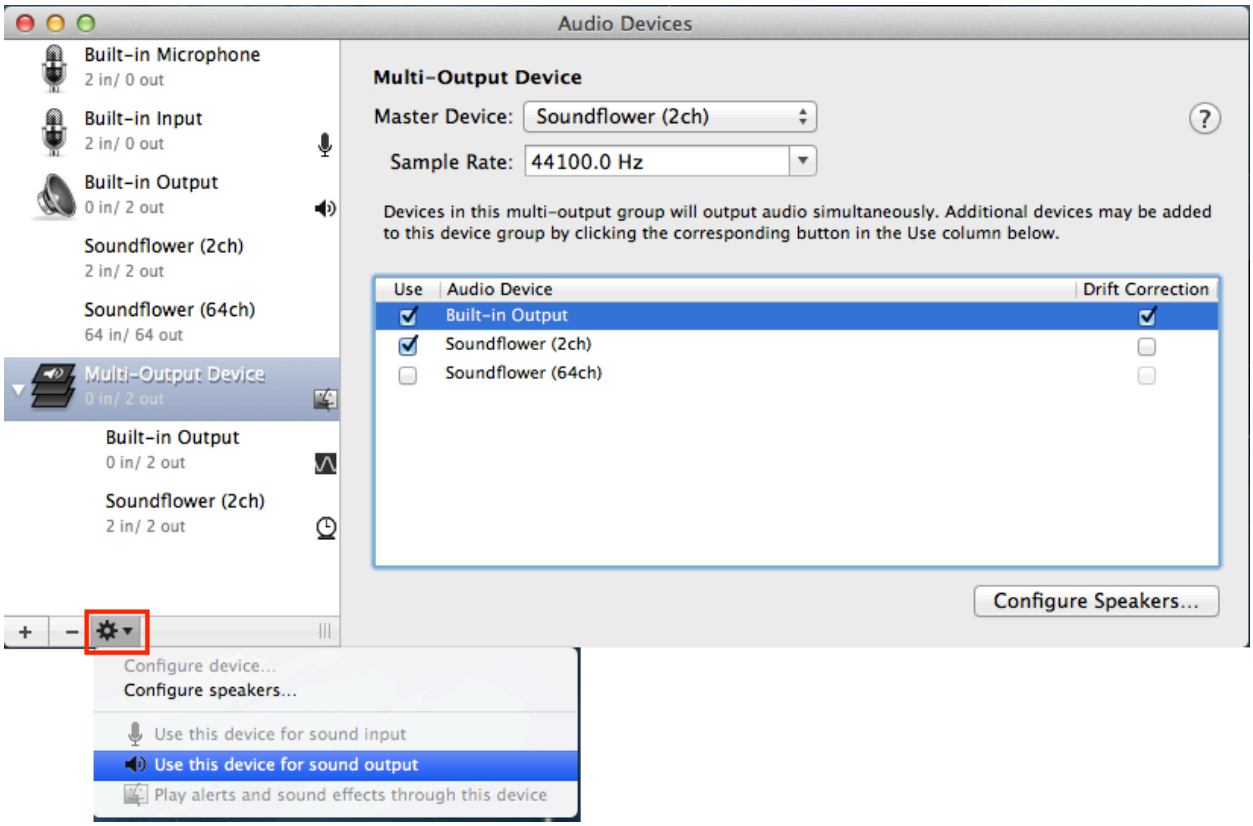
## Step 11 Display the real time FFT in Signal Scope

First Open /Applications/Audio MIDI Setup. Use the Window menu to display the Audio Devices window. We're going to create a Multiple Output Device so we can send ChucK audio through SoundFlower to SignalScope and be able to hear the audio at the same time.

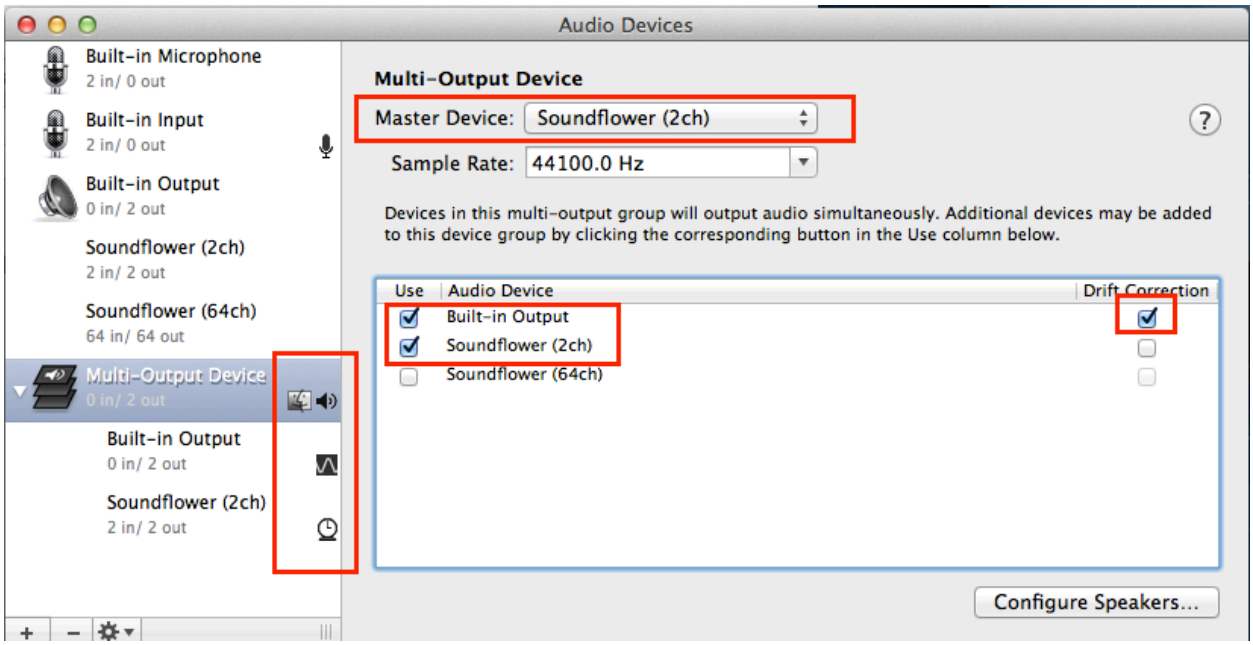
Click the plus sign at the bottom of the Audio Devices window and choose Create Multi-Output Device.



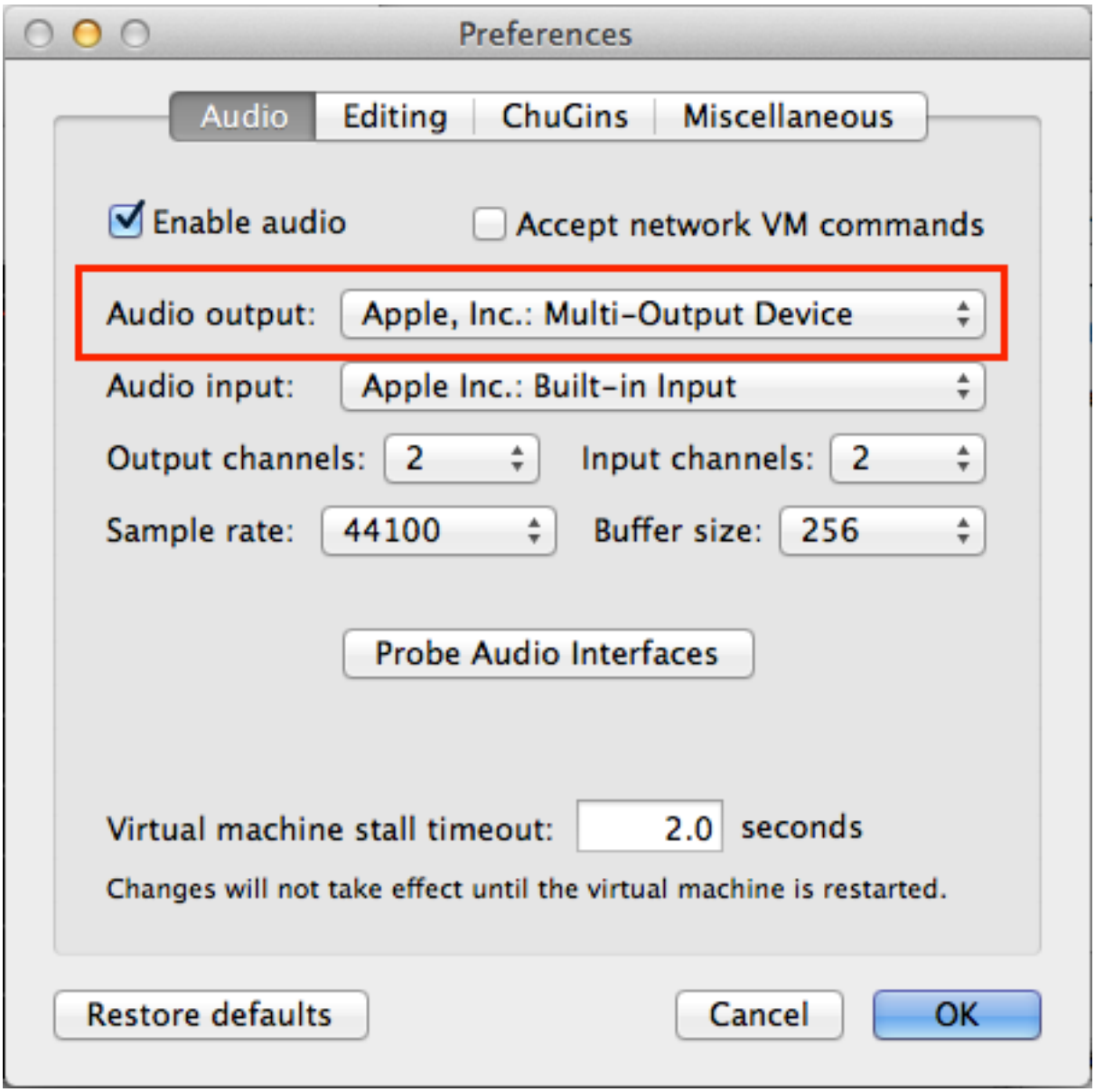
Next Select the Multi-Output Device in the left panel and click the pop-up menu that looks like a gear and choose Use this device for sound input.



Choose Soundflower 2(ch) as the Master device and check the Built-in Output and Soundflower as the Audio devices. Check Drift Compensation for Built-in Output.




Open miniAudicle Preferences and choose the Multi-Ouput Device for Audio output.




Quit and restart miniAudicle.

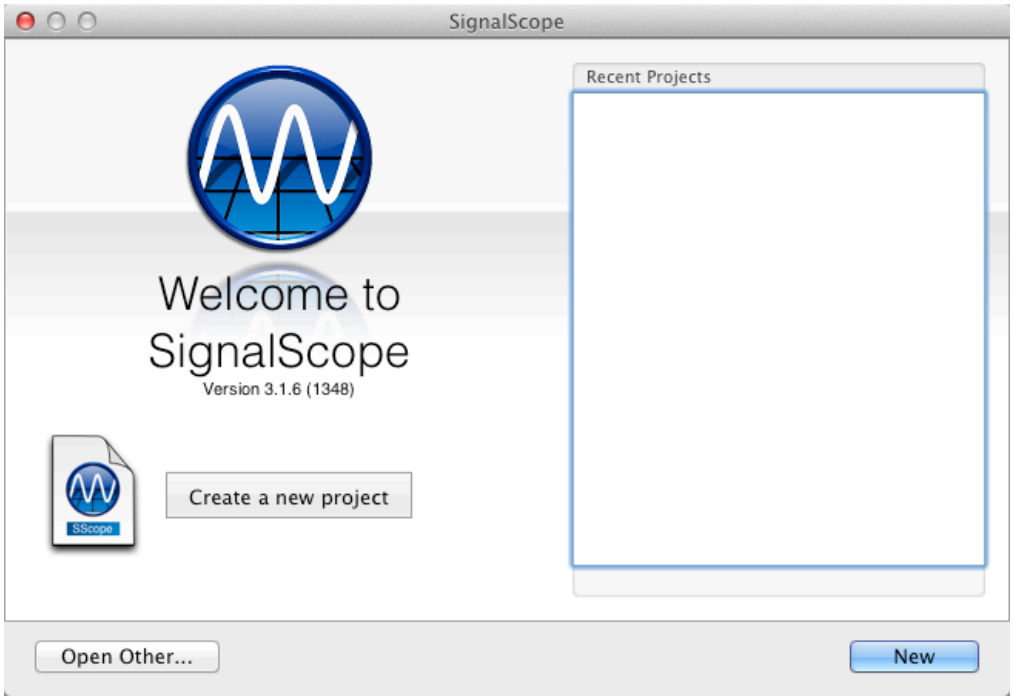
Run the Processing GUI.

Run simpleFIR.ck.

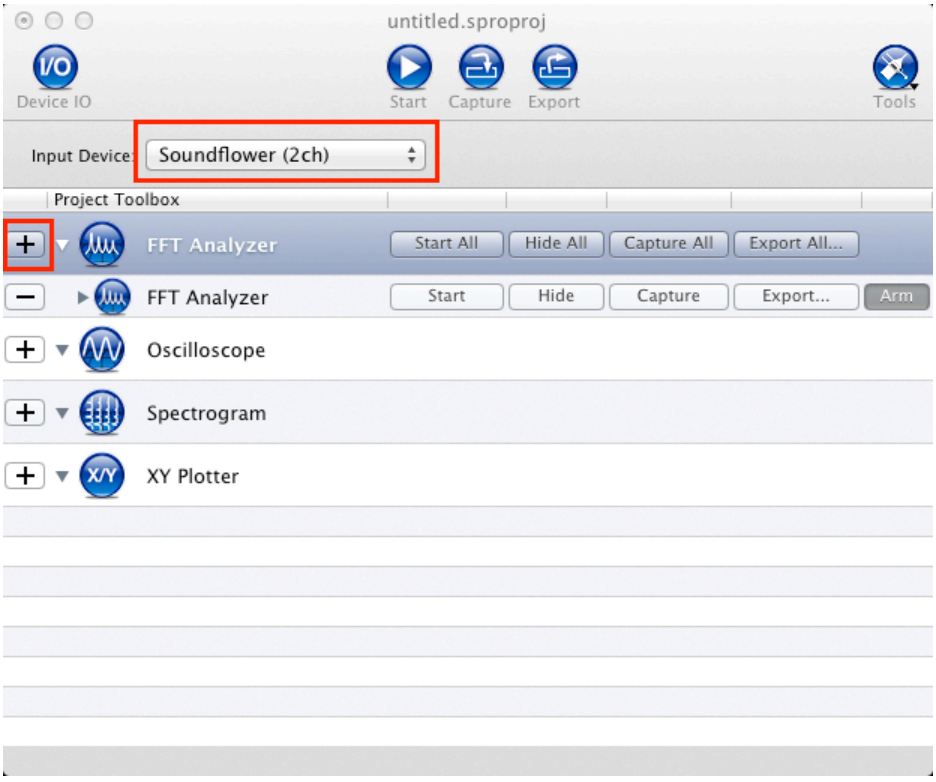
## Open /Applications/SignalScope

Click the Try It button

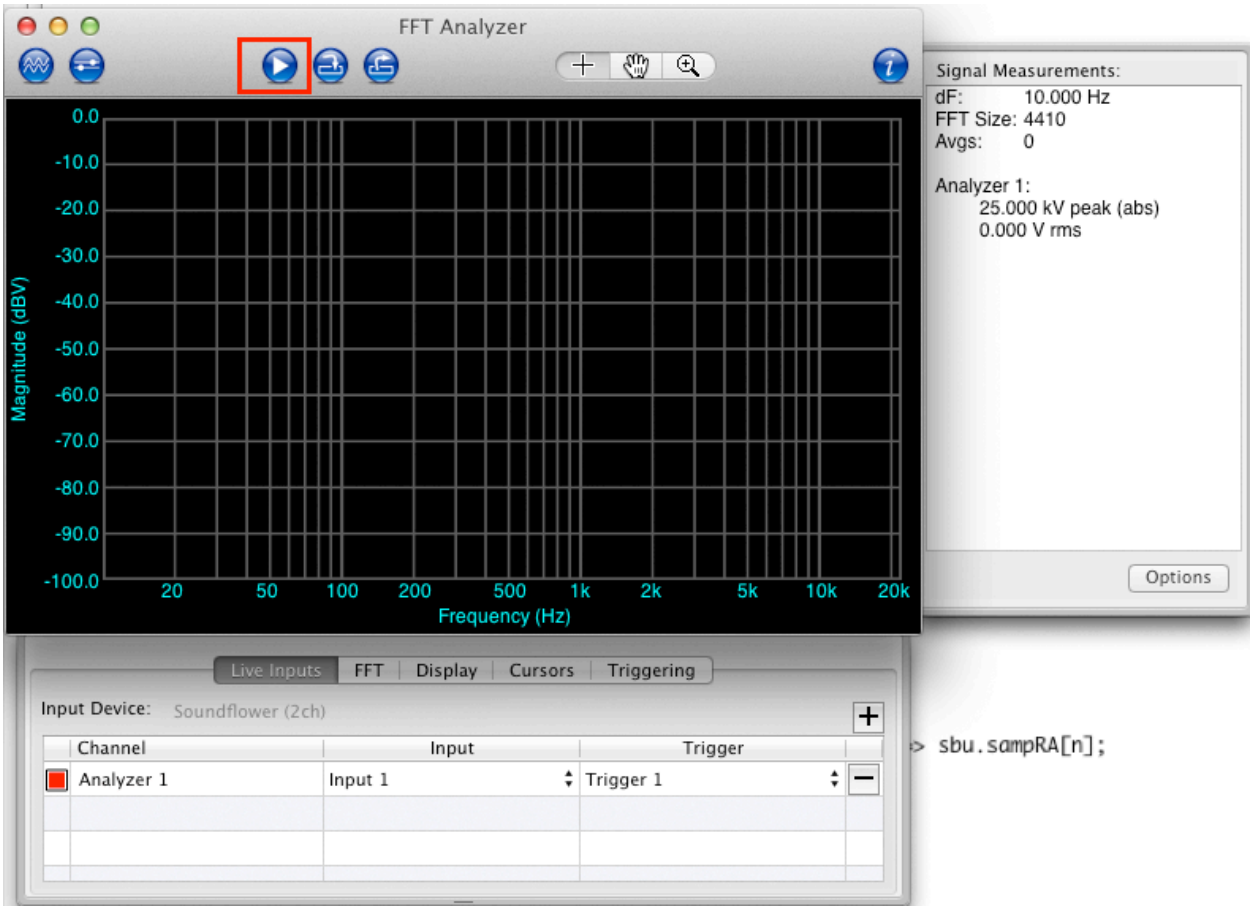
Click the New button



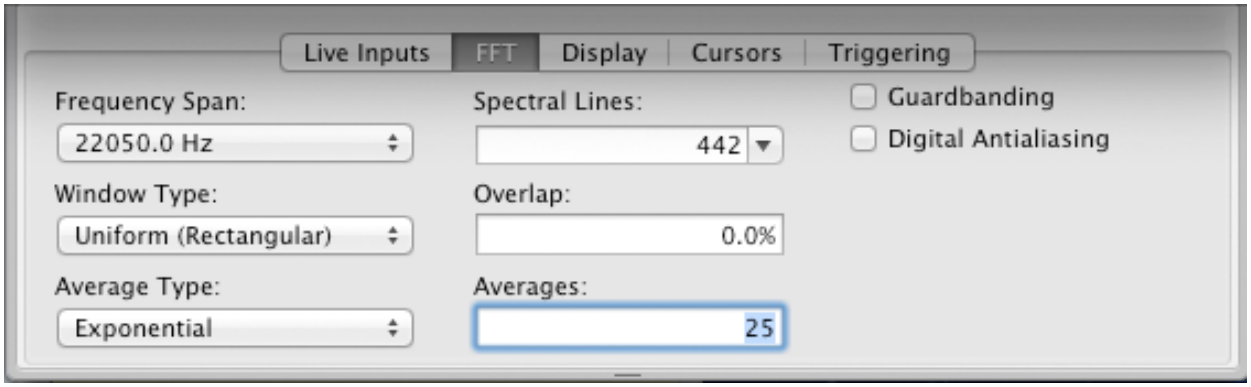

Click the Plus sign to the left of FFT Analizer and choose Soundflower as the input device.



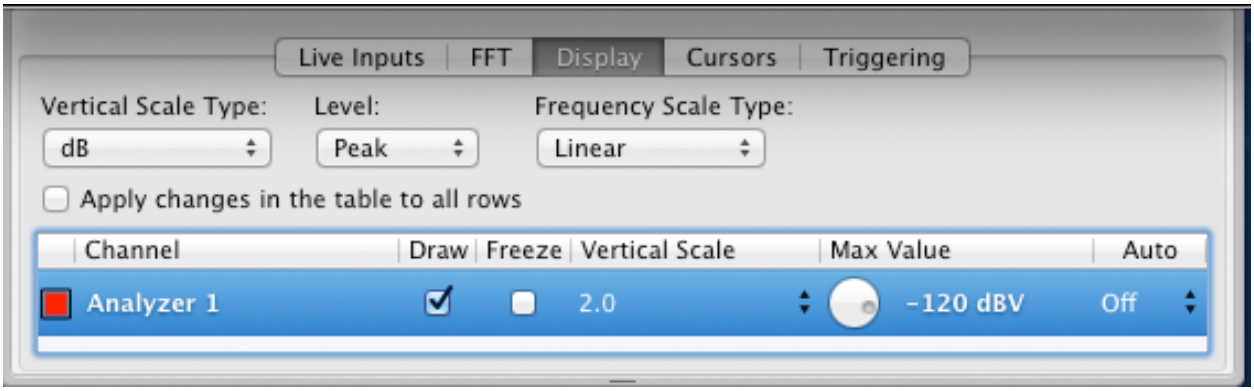
The FFT Analyzer window will appear. Click the Run button.



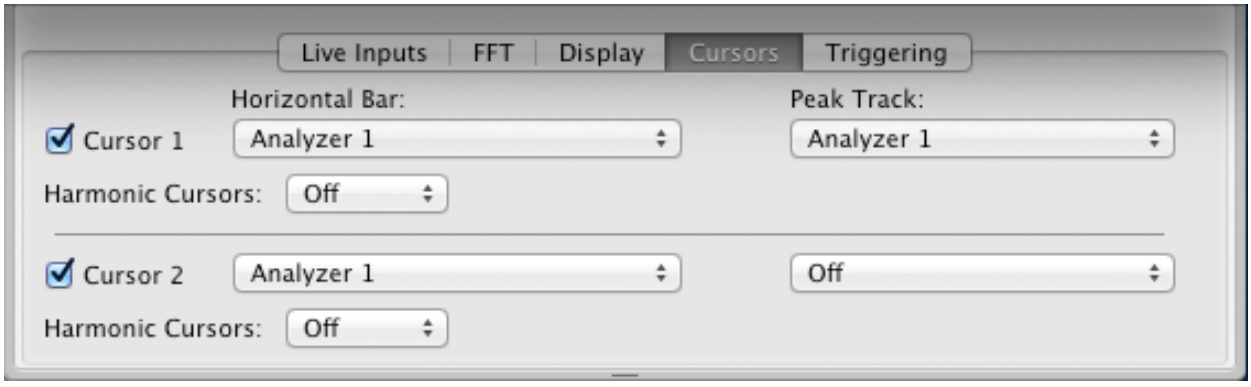

FFT Tab



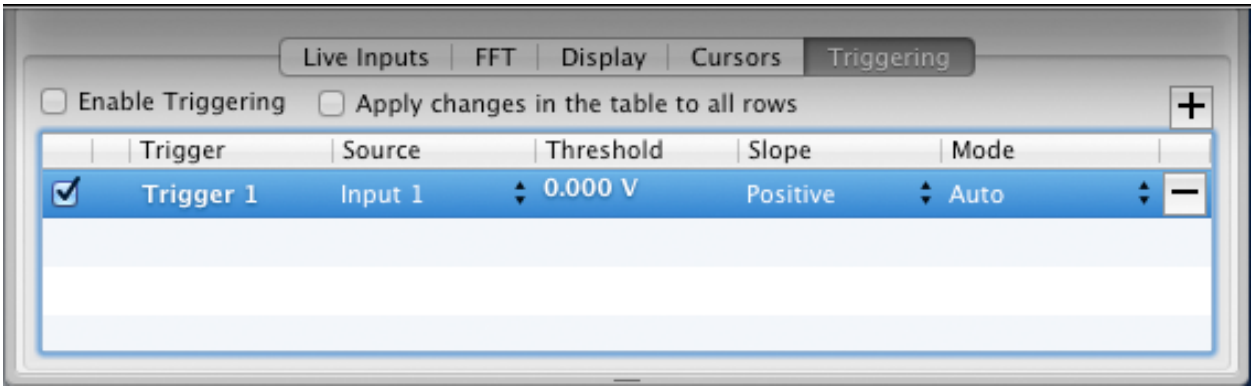
## Display Tab



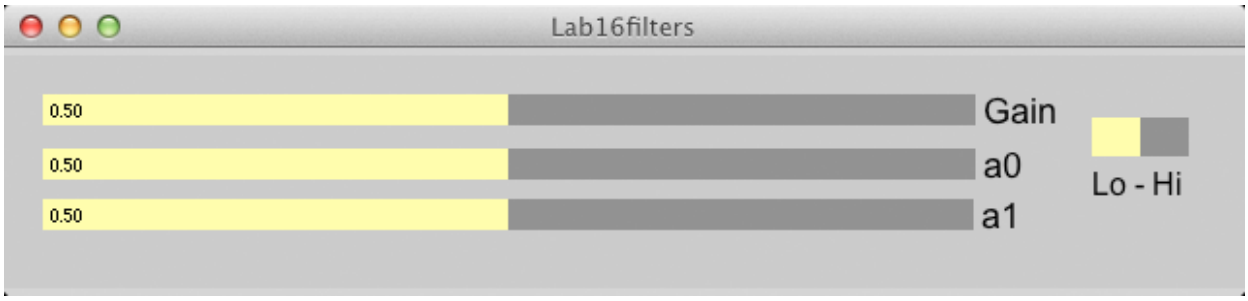

## Cursors Tab



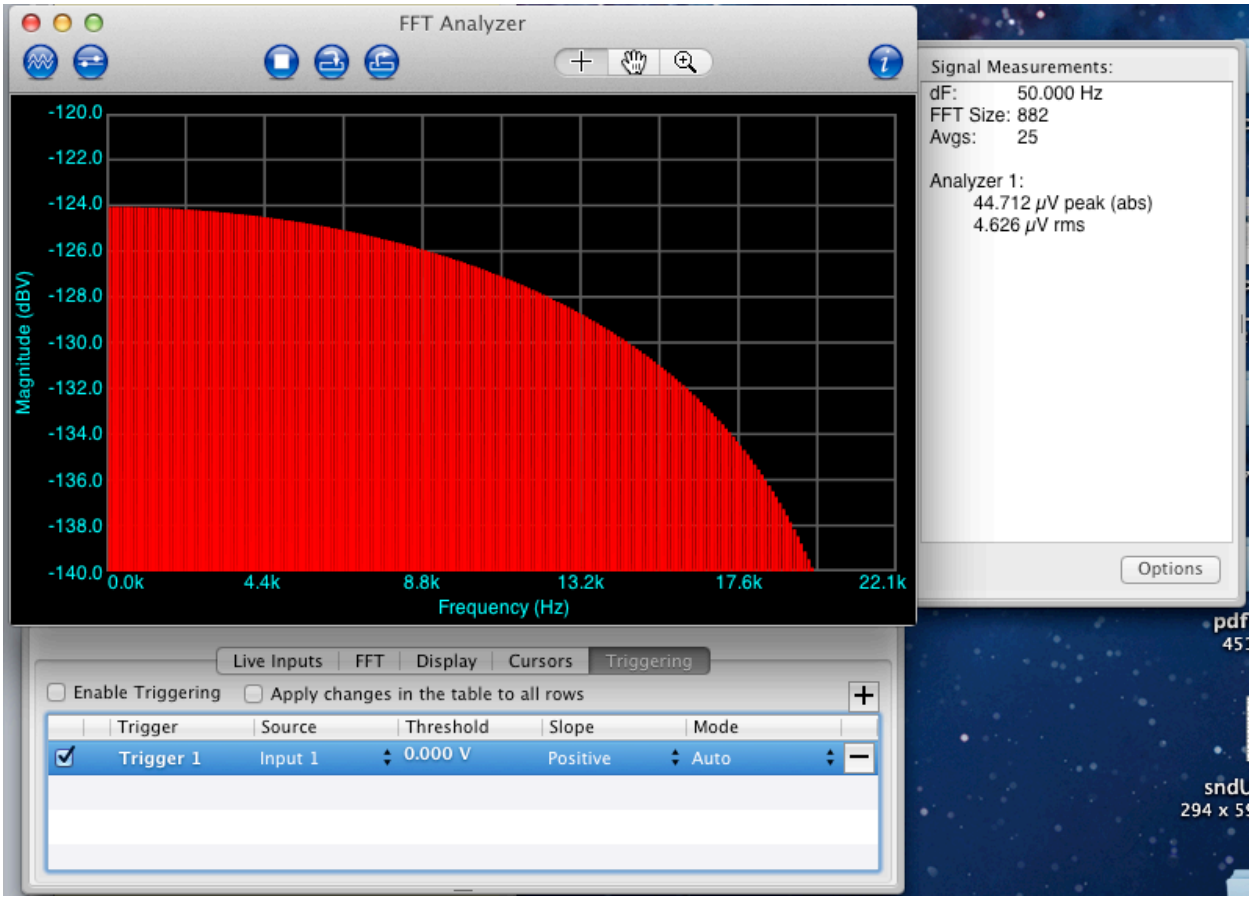

## Triggering Tab



GUI Settings 0.5 0.5 0.5

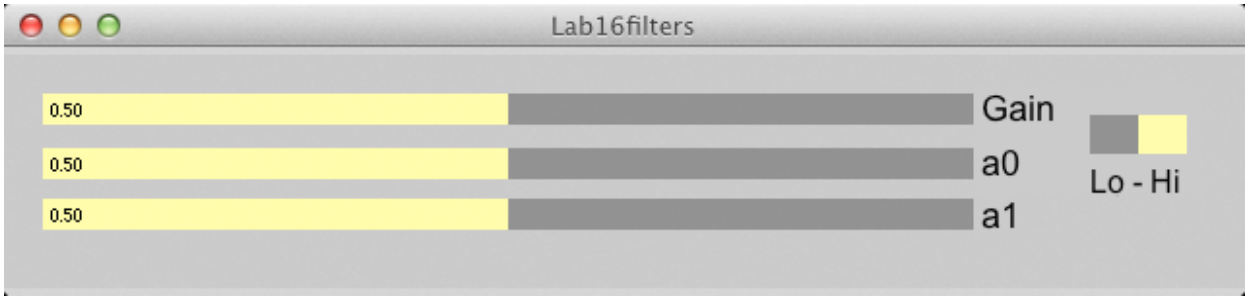



Freeze or Stop Low Pass Picture

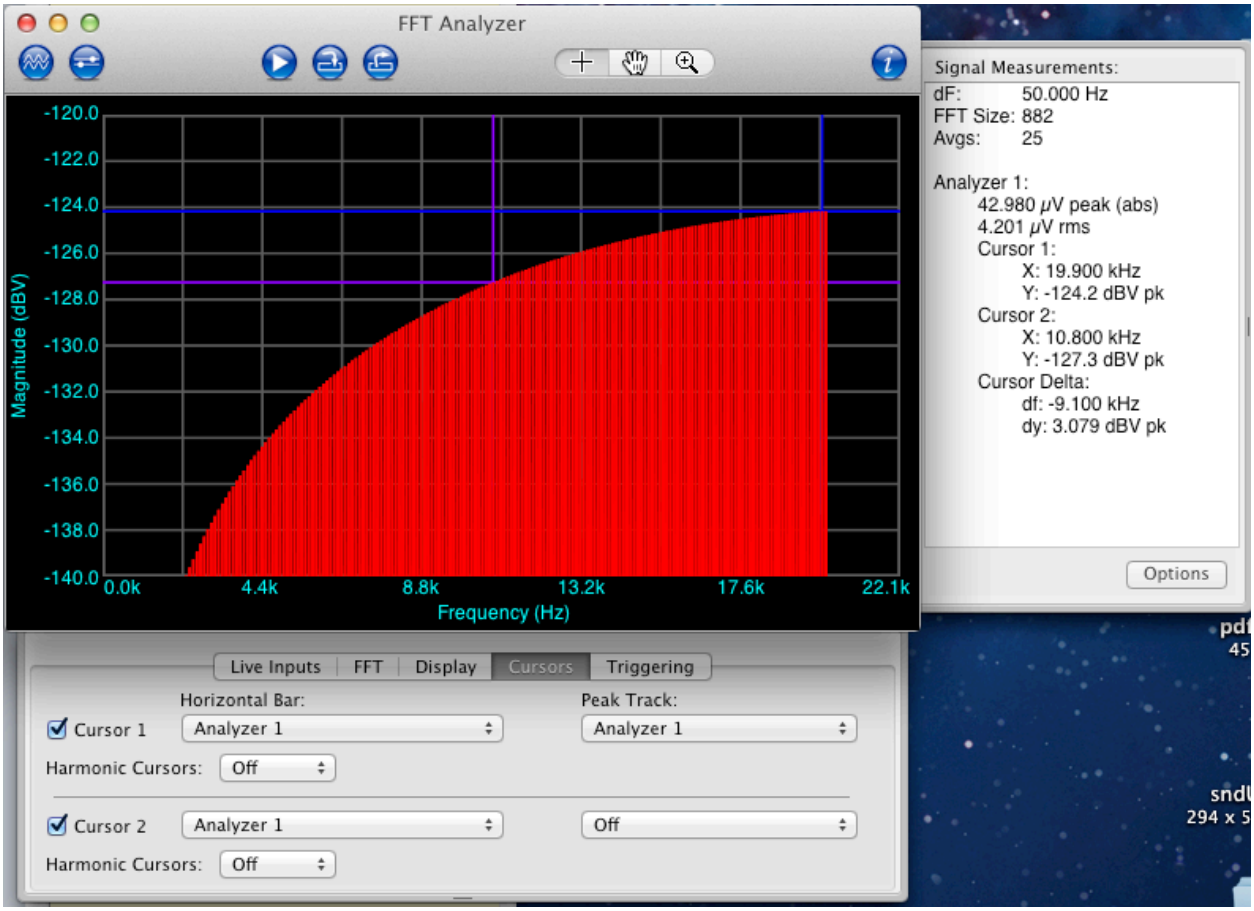


## High Pass

Settings 0.5 0.5 0.5

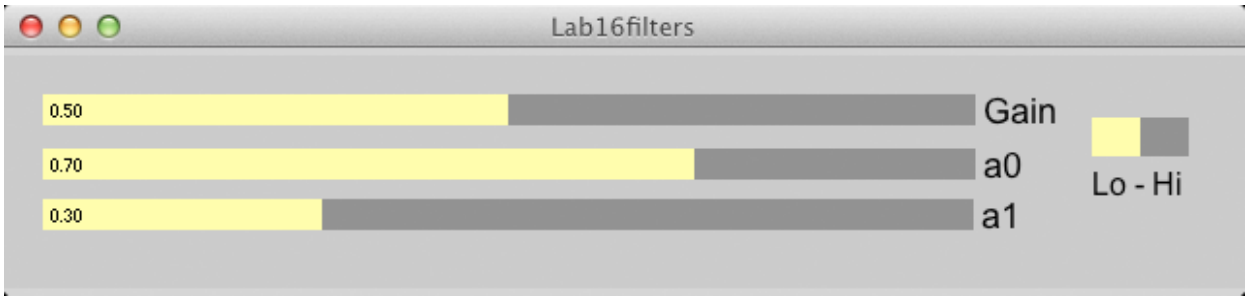


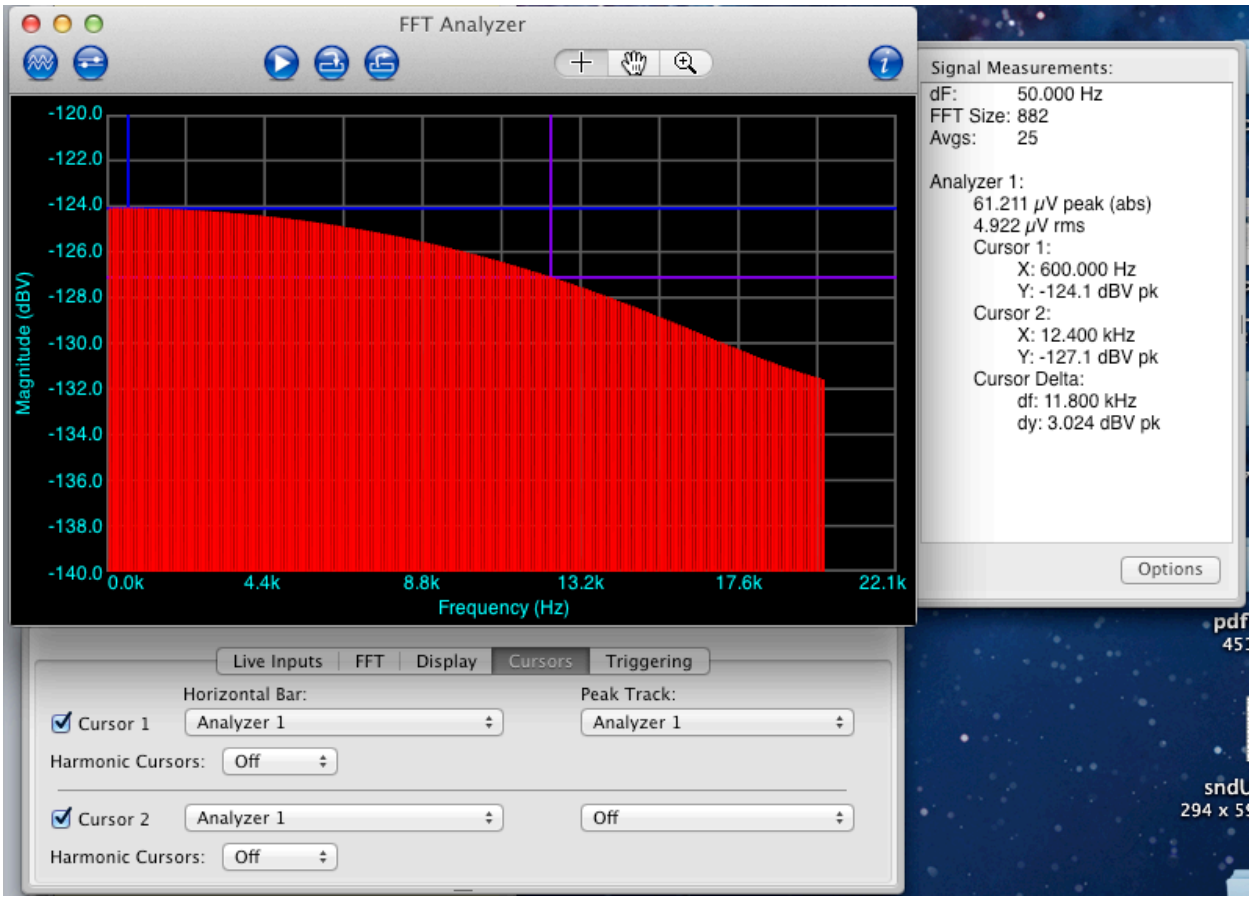

High Pass Picture



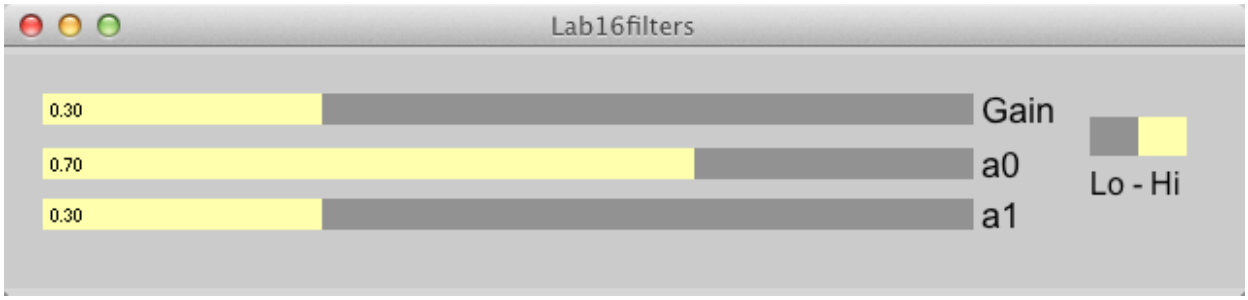

Experiment with other a0 and a1 settings.

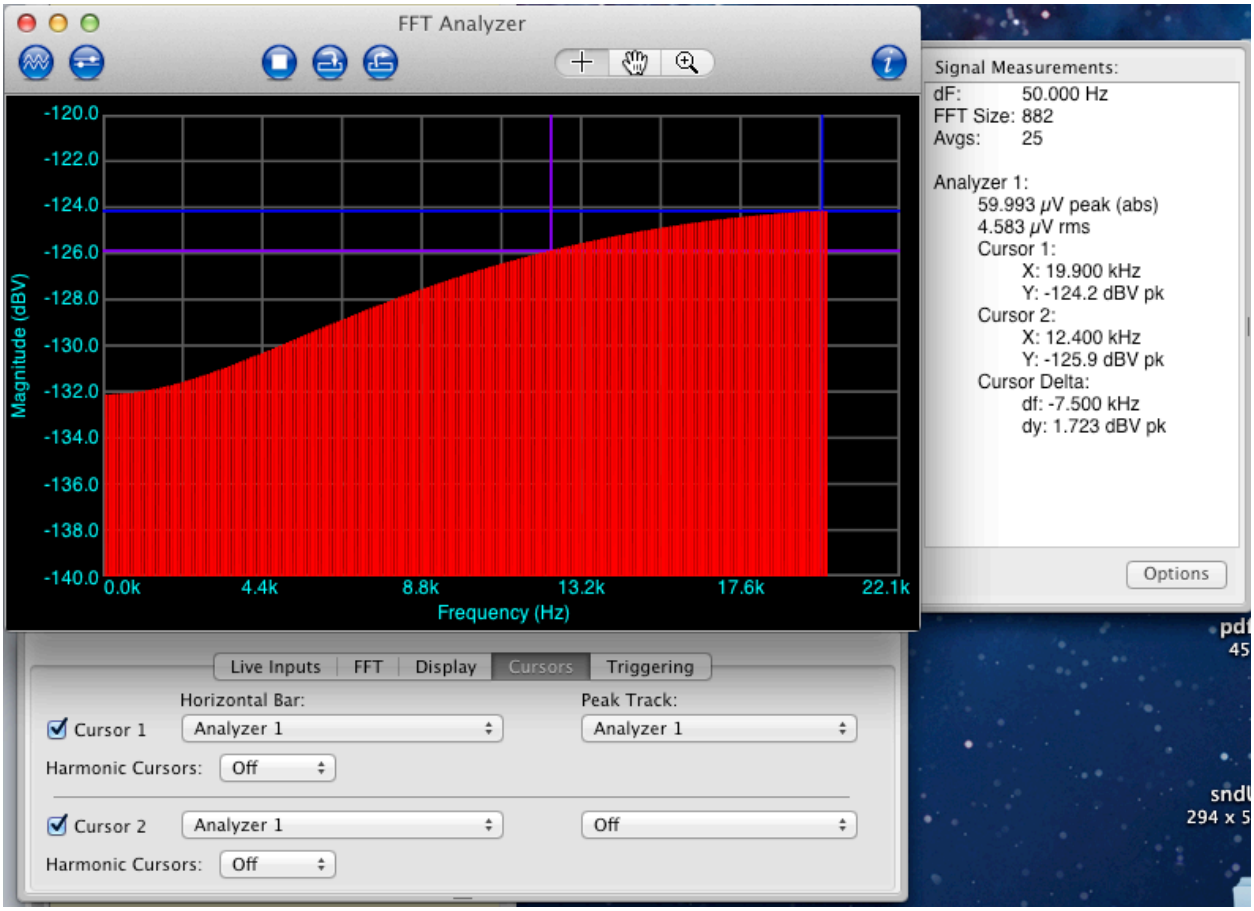
## Group 70 30




High Pass




# Using OSC in ChucK

http://booki.flossmanuals.net/chuck/_full/

by Rebecca Fiebrink

## To send OSC

Host Decide on a host to send the messages to. E.g., ”splash.local” if

sending to computer named ”Splash,” or ”localhost” to send to the same machine that is sending.

Port Decide on a port to which the messages will be sent. This is an integer, like 1234.

Message ”address” For each type of message you’re sending, decide on a way to identify this type of message, formatted like a web URL e.g., ”conductor/downbeat/beat1” or ”Rebecca/message1”

Message contents Decide on whether the message will contain data, which can be 0 or more ints, floats, strings, or any combination of them.

To set up a OSC sender in ChucK you'll need code like the following:

```
//Create an OscSend object:
OscSend xmit;
//Set the host and port of this object:
xmit.setHost("localhost", 1234);
```

For every message you want to send, start the message by supplying the address and format of contents, where ”f” stands for float, ”i” stands for int, and ”s” stands for string:

```
//To send a message with no contents:
xmit.startMsg("conductor/downbeat");
//To send a message with one integer:
xmit.startMsg("conductor/downbeat, i");
//To send a message with a float, an int, and another float:
xmit.startMsg("conductor/downbeat, f, i, f");
```

For every piece of information in the contents of each message, add this information to the message:

```
//to add an int:
xmit.addInt(10);
//to add a float:
xmit.addFloat(10.);
//to add a string:
xmit.addString("abc");
```

Once all parts of the message have been added, the message will

automatically be sent.

## To receive OSC

Decide what port to listen on. This must be the same as the port number of the sender(s) you want to listener to receive messages from. Message address and format of contents: This must also be the same as what the sender is using; i.e., the same as in the sender's startMsg function.

The following code shows how to setting up an OSC receiver with ChucK.

```
//Create an OscRecv object:
OscRecv orec;
//Tell the OscRecv object the port:
1234 => orec.port;
//Tell the OscRecv object to start listening for OSC messages on
that port:
orec.listen();
```

For each type of message, create an event that will be used to wait on that type of message, using the same argument as the sender's startMsg function:

```
orec.event("conductor/downbeat, i") @=> OscEvent myDownbeat;
```

To wait on an OSC message that matches the message type used for a particular event e, do

```
e => now;
```

This is just like waiting for regular Events in ChucK.

To process the message first it's necessary to grab the message out of the queue. In our example this can be achieved using e.nextMsg(). After we called this, we can use other methods on e to get the information we're interested in out of the message. We must call these functions in order, according to the formatting string we set up above.

```
e.getInt() => int i;
e.getFloat() => float f;
e.getString() => string s;
```

If you expect you may receive more than one message for an event at once,

you should process every message waiting in the cue:

```
while (e.nextMsg() != 0) {
  //process message here (no need to call nextMsg again
}
```