MUSC 208 Winter 2014
John Ellinger Carleton College

Lab 18 Delay Lines

Setup

Download the m208Lab18.zip files and move the folder to your desktop.

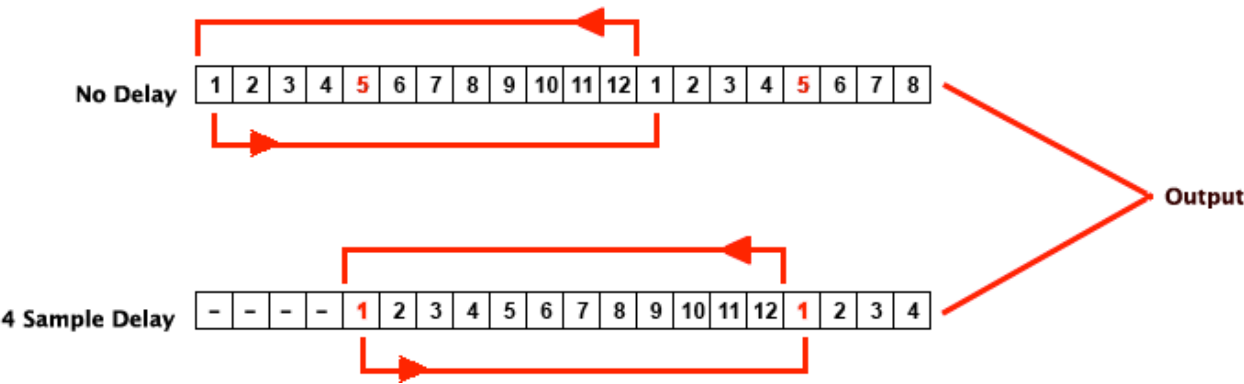
Delay Lines

Delay Lines are frequently used in audio software. The technique originated back in the days of the tape recorder where the tape was sometimes run through two tape decks whose play heads were a separated by several inches to several feet. When the sound passed the first play head it was heard through the speakers, and when it passed the second play head it was heard again delayed in time by the distance between the two play heads. If the distance was short the delay "fattened" the sound. If the delay was long the second sound was like an echo. In digital audio the delay results from playing a sound wave with a delayed copy of itself. In the digital audio domain a delay line is created by playing the original samples along with a delayed version of itself.

Recirculating Delay Line Code Overview

Create a new new file in miniAudicle and save it as circularDelayExample.ck. This example will be used to illustrate the principles of a recirculating delay line. Our example will consist of a twelve element array of samples whose values are the numbers 1-12. The samples will begin at one and start over from one (recirculate) when the array index reaches twelve. The delay amount will be four samples; i.e. the delay line will begin at one when the original samples reach five. The delay line will recirculate when the delay line index reaches twelve. The original sample and delayed sample are added for each output.

The following diagram illustrates what happens.



You'll need keep track of several variables: the total number of samples in the array, the number of samples to delay by, the current index position for both the non delayed line and the delayed line, and the current sample value of each line.

Enter and run this code.

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

// pretend these are sound samples
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] @=> int buf[];
buf.cap() => int bufLength;

// number of samples to delay by
4 => int dlyLength;

// sound starting index
0 => int bufIndex;
int bufVal; // snd sample value

// delay (dly) starting index
-dlyLength => int dlyIndex;
int dlyVal; // dly sample value

// number of times to repeat
3 => int numRepeats;
```

```
// main loop will run through numRepeats repeats
while (true)
{
    // check if we need to wrap back to the first sample
    // decrement the number of repeats
    if ( bufIndex >= bufLength )
    {
        0 => bufIndex; // reset
        numRepeats--; // decrement
        <<< "-----------", "" >>>;
        if ( numRepeats == 0 )
            me.exit();
    }

    // check if the delay needs to wrap back to the first sample
    if ( dlyIndex >= bufLength )
        0 => dlyIndex; // reset

    // sound starts right away
    buf[ bufIndex ] => bufVal;

    // delay starts when dlyIndex reaches 0
    if ( dlyIndex >= 0 )
        buf[dlyIndex] => dlyVal;

    // verify delay and number of repeats
    <<< bufVal, "\t", dlyVal >>>;

    // increment bufIndex and dlyIndex indices
    bufIndex++;
    dlyIndex++;
}
```

The output should look like this.

```
[chuck](VM): sporking incoming shred: 1 (circularDelayExample.ck)...
1    0
2    0
3    0
4    0
5    1
6    2
7    3
8    4
9    5
10   6
11   7
12   8
-----------
1    9
2    10
3    11
4    12
5    1
6    2
7    3
8    4
```

This is the same principle used to create audio delay lines except we'll be working with 44,100 samples per second.

Delay Types

Delay times can be measured in samples, but they're usually measured in milliseconds or seconds. Tempo delays are calculated as a fractions of a beat.

Delay times can be categorized as:

- Short Delays - generally less than 10 ms
- Medium Delays - 10 - 100 ms
- Long Delays - greater than 100 ms

Short Delays

Short delays range from a few samples to a few milliseconds, generally less than 10 ms and are sometimes used to compensate for phase issues in stereo recordings. If two microphones are more than a few inches apart, sounds reach the left and right microphones at different times and can cause phase

cancellation. By calculating the distance between the microphones and knowing the speed of sound, you can calculate the number of samples needed to delay one of the signals and reduce phase cancellation problems.

Medium Delays

Medium delays in the rage of 10-50 milliseconds have the effect of "fattening" a signal. Medium delays in the range of 60-100 milliseconds are sometimes called Slapback echoes.

Long Delays

Delays longer than 100 milliseconds are heard as distinct echoes.

delayLine.ck

The delayLine.ck program can be used to test different delay times.

```
// delayLine.ck
// John Ellinger Music 208 Winter2014
/*
these sound files are found in the /wav folder
the names can be copied and pasted into the code below
ViolinPizzicato.wav
CelloSolo.wav
music208.wav
*/

Impulse imp => dac;
SndBuf buf;
buf.read( me.sourceDir() + "/wav/ViolinPizzicato.wav" );
buf.samples() => int bufLength;

// specify delay time in milliseconds and convert to samples
( 50::ms/samp ) $ int  => int dlyLength; // ms to samples

// sound starting index
0 => int bufIndex;
0.0 => float bufVal; // snd sample value
```

```
// delay starting index
-dlyLength => int dlyIndex;
0.0 => float dlyVal; // delay sample value

// number of times to reapeat
2 => int numRepeats;

while ( true )
{
    // check we need to wrap around to the first sample
    // decrement the number of repeats
    if ( bufIndex >= bufLength )
    {
        0 => bufIndex; // reset
        numRepeats--; // decrement
        if ( numRepeats == 0 )
            me.exit();
    }

    // check if the delay needs to wrap back to the first sample
    if ( dlyIndex >= bufLength )
        0 => dlyIndex; // reset

    // sound starts right away
    buf.valueAt( bufIndex ) => bufVal;

    // delay starts when dlyIndex reaches 0
    if ( dlyIndex >= 0 )
        buf.valueAt( dlyIndex ) => dlyVal;

    // play em
    bufVal + dlyVal => imp.next;
    1::samp => now;

    // increment bufIndex and dlyIndex indices
    bufIndex++;
    dlyIndex++;

}
```

Test each wav file using these delays times

The ViolinPizzicato.wav file

Short: < 10 ms
Medium: 10 - 50 ms
Slapback: 60 – 100 ms
Long: > 100 ms

The ViolinPizzicato.wav file uses a violin technique where the string is plucked instead of bowed. Pizzicato sounds have a very sharp attack and almost no sustain so the delay can be clearly heard, even at short delays. The cello sounds have a smooth attack and long sustain making it hard to hear short delays. The third sound is speech, the all too familiar music208.wav file.

I wrote this melody several years ago for testing in MUSC 108 class.

Play The CelloSolo.wav As A Round

The example melody was designed to play as a round. The tempo is 120 and a quarter note lasts for 500 ms. Use the CelloSolo.wav with a delay of 3000 ms and you'll hear the round. Modify the code to repeat the melody exactly three times.

Multi-tap Delay

It's possible to have more than one delay line playing at a time. This is called a multi-tap delay. Each delay position is called a tap and can be arbitrarily placed. The multiTapDelay.ck example expands delayLine.ck to create three taps.

```
// multiTapDelay.ck
// John Ellinger Music 208 Winter2014
/*
these sound files are found in the /wav folder
the names can be copied and pasted into the code below
ViolinPizzicato.wav
CelloSolo.wav
music208.wav
*/

Impulse imp => dac;
SndBuf buf;
me.sourceDir() + "/wav/ViolinPizzicato.wav" => buf.read;
buf.samples() => int bufLength;

// sound starting index
0 => int bufIndex;
0.0 => float bufVal;

( 100::ms/samp ) $ int => int dlyLength1;
-dlyLength1 => int dlyIndex1;
0.0 => float dlyVal1; // delay1 sample value

( 300::ms/samp ) $ int  => int dlyLength2;
-dlyLength2 => int dlyIndex2;
0.0 => float dlyVal2; // delay2 sample value

( 600::ms/samp ) $ int  => int dlyLength3;
-dlyLength3 => int dlyIndex3;
0.0 => float dlyVal3; // delay3 sample value

// Amplitude
0.6 => float A0;
0.25 => float A1;
0.25 => float A2;
0.6 => float A3;

// another way to do repeats
2 => int numRepeats;
now +  numRepeats * buf.length() => time later;
```

```
while ( now < later )
{
    // check we need to wrap around to the first sample
    // decrement the number of repeats
    if ( bufIndex >= bufLength )
        0 => bufIndex; // reset

    // check if the delays needs to wrap back to the first sample
    if ( dlyIndex1 >= bufLength )
        0 => dlyIndex1; // reset
    if ( dlyIndex2 >= bufLength )
        0 => dlyIndex2; // reset
    if ( dlyIndex3 >= bufLength )
        0 => dlyIndex3; // reset

    // sound starts right away
    buf.valueAt( bufIndex ) => bufVal;

    // delay starts when dlyIndex reaches 0
    if ( dlyIndex1 >= 0 )
        buf.valueAt( dlyIndex1 ) => dlyVal1;
    if ( dlyIndex2 >= 0 )
        buf.valueAt( dlyIndex2 ) => dlyVal2;
    if ( dlyIndex3 >= 0 )
        buf.valueAt( dlyIndex3 ) => dlyVal3;

    // play em
    A0*bufVal + A1*dlyVal1 + A2*dlyVal2 + A3*dlyVal3 => imp.next;
    1::samp => now;

    // increment bufIndex and dlyIndex indices
    bufIndex++;
    dlyIndex1++;
    dlyIndex2++;
    dlyIndex3++;
}
```

Experiment with different delays and amplitudes.

Canyon Echo

Use multiTapDelay.ck to create an echo effect with the music208.wav file using long delays and decreasing amplitudes for each echo. Try these settings.

```
( 500::ms/samp ) $ int => int dlyLength1;
( 750::ms/samp ) $ int  => int dlyLength2;
( 900::ms/samp ) $ int  => int dlyLength3;

// Amplitude
0.4 => float A0;
0.1 => float A1;
0.05 => float A2;
0.001=> float A3;
```

Tempo Sync Multi-tap Delay

If the multiple delays follow the tempo they are known as Tempo Sync Delays. Create and run this code.

```
// multiTapTempoSync.ck
// John Ellinger Music 208 Winter2014
Impulse imp => dac;
SndBuf buf;
me.sourceDir() + "/wav/ViolinPizzicato.wav" => buf.read;
buf.samples() => int bufLength;
0 => int bufIndex;

4 => int NUMTAPS;
int delayLength[ NUMTAPS ];
float amp[ NUMTAPS ];

function void initTap( int n, float msTime )
{
    -( msTime::ms/samp ) $ int @=> delayLength[n];
}

// Tempo of original melody is 120 bpm
500.0 / 4 => float t16; // sixteenth note
-1000000 => float rest; // trick to create a rest
```

```
initTap( 0, 0 * t16);
initTap( 1, 1 * t16);
initTap( 2, 2 * t16 );
initTap( 3, 3 * t16 );

0.7 @=> amp[ 0 ];
0.3 @=> amp[ 1 ];
0.5 @=> amp[ 2 ];
0.3 @=> amp[ 3 ];

float bufOut;
float tapOut;

// 2 repeats
now +  2 * buf.length() => time later;
while ( now < later )
{
    0 => float tapOut;
    for ( 0 => int n; n < NUMTAPS; n++ )
    {
        if ( delayLength[n] < 0 )
            0 => tapOut;
        else
            tapOut+amp[n]*buf.valueAt(delayLength[n]) => tapOut;

        delayLength[n]++;
        if ( delayLength[n] >= bufLength ) // check wrap around
            0 => delayLength[n];
    }

    bufIndex++;
    if ( bufIndex >= bufLength ) // check wrap around
        0 => bufIndex;

    tapOut => imp.next;
    1::samp => now;
}
```

Test these rhythms using the following settings.

```
initTap( 0, 0 * t16);
initTap( 1, 1 * t16);
initTap( 2, rest );
initTap( 3, rest );
```

```
initTap( 0, 0 * t16);
initTap( 1, 2 * t16);
initTap( 2, rest );
initTap( 3, rest );
```

```
initTap( 0, 0 * t16);
initTap( 1, 3 * t16);
initTap( 2, rest );
initTap( 3, rest );
```

```
initTap( 0, 0 * t16);
initTap( 1, 1 * t16);
initTap( 2, 2 * t16 );
initTap( 3, rest );
```

```
initTap( 0, 0 * t16);
initTap( 1, 1 * t16);
initTap( 2, 3 * t16 );
initTap( 3, rest );
```

```
initTap( 0, 0 * t16);
initTap( 1, 2 * t16);
initTap( 2, 3 * t16 );
initTap( 3, rest );
```

```
initTap( 0, 1 * t16);
initTap( 1, 2 * t16);
initTap( 2, 3 * t16 );
initTap( 3, rest );
```

``` initTap(0, 0 * t16); initTap(1, 1 * t16); initTap(2, 2 * t16); initTap(3, 3 * t16); ```	

```
initTap( 0, 0 * t16);
initTap( 1, 1 * t16);
initTap( 2, 2 * t16 );
initTap( 3, 3 * t16 );
```

tinySlices.ck

tinySlices.ck uses a delay line to hold a a small slice of sound, 100 ms duration or 441 samples. The sound file is cut up into sequential slices of 441 samples, then each slice is played 15 times before moving to the next slice. Enter and run this code.

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

Impulse imp => dac;
SndBuf buf;
me.sourceDir() + "/wav/music208new.wav" => buf.read;
buf.samples() => int bufLen;

// play original wav file
0 => buf.pos;
buf => dac;
buf.length() => now;
buf =< dac;
1::second => now; // pause

441  => int sliceLength; // 100 ms
float delayLine[ sliceLength ];
(bufLen / sliceLength) $ int => int numSlices;
15 => int numRepeats;

function void fillDelayLine( int start, int sliceLength )
{
    sliceLength => delayLine.size; // dynamic resize
    for ( 0 => int n; n < sliceLength; n++ )
        buf.valueAt( n + start ) => delayLine[n];
}
```

```
int sliceBegin;
int sliceEnd;
for ( 0 => int n; n < numSlices; n++ )
{
    n*sliceLength  => sliceBegin;
    (n+1)*sliceLength - 1 => sliceEnd;
    fillDelayLine( sliceBegin, sliceLength );

    for ( 0 => int r; r < numRepeats; r++ )
    {
        for ( 0 => int d; d < delayLine.cap(); d++ )
        {
            delayLine[ d ] => imp.next;
            1::samp => now;
        }
    }
}
```

Add these two lines to randomize the sliceLength and the number of repeats.

```
for ( 0 => int n; n < numSlices; n++ )
{
    Math.random2( 60, 600 ) => sliceLength;
    n*sliceLength  => sliceBegin;
    (n+1)*sliceLength - 1 => sliceEnd;
    fillDelayLine( sliceBegin, sliceLength );
    Math.random2( 10, 30 ) => numRepeats;

    for ( 0 => int r; r < numRepeats; r++ )
    {
        for ( 0 => int d; d < delayLine.cap(); d++ )
        {
            delayLine[ d ] => imp.next;
            1::samp => now;
        }
    }
}
```

You may notice a discontinuity or click between slices. You can smooth that out with an envelope or window. One of the easiest windows to use is half a sine wave. We've used this formula many times before to generate a sine wave.

$$y[n] = \sin\left(2 \bullet \pi \bullet Freq \bullet \frac{n}{SR}\right)$$

Use a half sine wave envelope to eliminate the discontinuities. By setting SR = sliceLength and Freq = 0.5 the formula becomes:

$$y[n] = \sin\left(2 \bullet \pi \bullet \frac{1}{2} \bullet \frac{n}{sliceLength}\right) = \sin\left(\frac{n \bullet \pi}{sliceLength}\right)$$

Modify the fillDelayLine() function. Save it as tinySlices3.ck and run the program.

```
function void fillDelayLine( int start, int sliceLength )
{
    sliceLength => delayLine.size; // dynamic resize
    for ( 0 => int n; n < sliceLength; n++ )
        buf.valueAt(n+start)*Math.sin(n*pi/sliceLength) => delayLine[n];
}
```

The Karplus-Strong Algorithm

The Karplus-Strong algorithm produces the sound of a plucked string. It's a very simple algorithm that was used in many early synthesizers. It starts by filling a delay line with random values (noise).

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

/*************** WARNING ***************
    LOUD!!!         LOUD!!!        LOUD!!!
**************** WARNING ***************/

// Fill a delay line with random noise and play it

Impulse imp => dac;
44100.0 => float SR;
// perceived pitch
220 => float freq;
SR/freq  => float samplesInOnePeriod;
samplesInOnePeriod $ int => int DELAYLENGTH;
float ks[ DELAYLENGTH ];

// fill the delay line with random noise
for ( 0 => int n; n < DELAYLENGTH; n++ )
{
    Math.random2f( -1.0, 1.0 ) => ks[n];
}

int n;
now + 1::second => time later;
while (now < later )
{
    ks[n] => imp.next;
    n++;
    if ( n > DELAYLENGTH - 1 )
        0 => n;

    1::samp => now;
}
```

When you play it you'll hear a buzzy pitch. Let's low pass filter it using the moving average filter we learned about in Lab16 (simpleLowpass.ck).

Add/modify these lines

```
int n;
0.0 => float prevSamp;
now + 1::second => time later;
while (now < later )
{
    0.5 * ks[n] + 0.5 * prevSamp => ks[n];
    ks[n] => prevSamp;
    n++;
    if ( n > DELAYLENGTH - 1 )
        0 => n;

    ks[n] => imp.next;
    1::samp => now;
}
```

This should sound more like a plucked string although it doesn't decay very quickly as you can hear by changing

```
now + 1::second => time later;
```

to

```
now + 4::second => time later;
```

Let's add an amplitude decay factor.

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

Impulse imp => dac;
44100.0 => float SR;
// perceived pitch
220 => float freq;

// this is the preferred formula to get a better match
// to the actual frequency specified
( SR/freq - 0.5 )  => float samplesInOnePeriod;
samplesInOnePeriod $ int => int DELAYLENGTH;
float ks[ DELAYLENGTH ];

// fill the delay line with random noise
for ( 0 => int n; n < DELAYLENGTH; n++ )
{
    Math.random2f( -1.0, 1.0 ) => ks[n];
}

int n;
0.0 => float prevSamp;
0.997 => float ampDecay;
now + 4::second => time later;
while (now < later )
{
    ampDecay * ( 0.5 * ks[n] + 0.5 * prevSamp ) => ks[n];
    ks[n] => prevSamp;
    n++;
    if ( n > DELAYLENGTH - 1 )
        0 => n;

    ks[n] => imp.next;
    1::samp => now;
}
```

Experiment with different values for ampDecay.

```
0.9999 => float ampDecay;
0.999 => float ampDecay;
0.99 => float ampDecay;
0.9 => float ampDecay;
```

Let's create a KarplusStrong class. You can reuse most of the above code.

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

public class KarplusStrong
{
    function void pluck(  float freq )
    {
        // copied from karplusStrong3.ck
        Impulse imp => dac;
        44100.0 => float SR;
        // 220 => float freq;
        // this is the preferred formula
        ( SR / freq - 0.5 )  => float samplesInOnePeriod;
        samplesInOnePeriod $ int => int DELAYLENGTH;
        float ks[ DELAYLENGTH ];

        // use only the samples we need
        for ( 0 => int n; n < DELAYLENGTH; n++ )
        {
            Math.random2f( -1.0, 1.0 ) => ks[n];
        }

        int n;
        0.0 => float lastSamp;
        0.997 => float ampDecay;
        now + 4::second => time later;
```

```
        while (now < later )
        {
            ampDecay * ( 0.5*ks[n] + 0.5*lastSamp ) => ks[n];
            ks[n] => lastSamp;
            n++;
            if ( n > DELAYLENGTH - 1 )
                0 => n;

            ks[n] => imp.next;
            1::samp => now;
        }
    }
}
```

Create and save the karplusMelody.ck file.

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

// the class
KarplusStrong ks;

.5 => float wait;

spork ~ ks.pluck( Std.mtof(60) ); wait::second => now;
spork ~ ks.pluck( Std.mtof(62) ); wait::second => now;
spork ~ ks.pluck( Std.mtof(64) ); wait::second => now;
spork ~ ks.pluck( Std.mtof(65) ); wait::second => now;
spork ~ ks.pluck( Std.mtof(67) ); wait::second => now;
spork ~ ks.pluck( Std.mtof(69) ); wait::second => now;
spork ~ ks.pluck( Std.mtof(71) ); wait::second => now;
spork ~ ks.pluck( Std.mtof(72) ); wait::second => now;

spork ~ ks.pluck( Std.mtof(60) ); wait::second => now;
spork ~ ks.pluck( Std.mtof(55) ); wait::second/2 => now;
spork ~ ks.pluck( Std.mtof(55) ); wait::second/2 => now;
spork ~ ks.pluck( Std.mtof(56) ); wait::second => now;
spork ~ ks.pluck( Std.mtof(55) ); wait::second*2 => now;
spork ~ ks.pluck( Std.mtof(59) ); wait::second => now;
spork ~ ks.pluck( Std.mtof(60) ); wait::second*2 => now;
```

```
3 => wait;
spork ~ ks.pluck( Std.mtof(60) );
spork ~ ks.pluck(Std.mtof(64) );
spork ~ ks.pluck( Std.mtof(70) );

2::second => now;
```

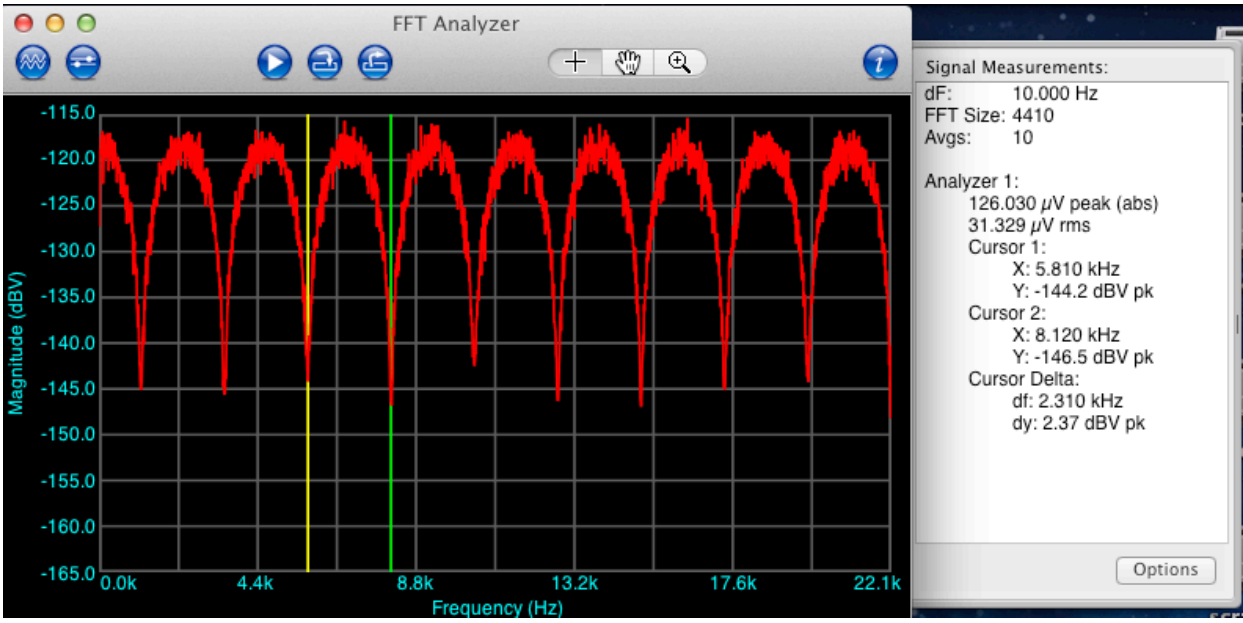
Create the playKarplusMelody.ck to load the class and play the melody.

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

Machine.add( me.sourceDir() + "/karplusStrongClass.ck" );
Machine.add( me.sourceDir() + "/karplusMelody.ck" );
```

Comb Filter

If you combine a delay line with a buffer of noise the FFT spectrum resembles the teeth of a comb which produces multiple bandreject filters at integer harmonics of the delay line fundamental frequency. This picture shows the SoundScope FFT analysis



The combFilter.ck is ready to run in the m208Lab18 download folder. But don't bother running it just now.

Hum Removal

One common use of a comb filter is removing 60 Hz hum that is sometimes picked up from household electricity.

Run combFilter_RemoveHum.ck found in the download folder. You'll first hear the StarsNStripesHum.wav file with an audible 60 Hz hum. Actually I mixed the original wave file with a 60 Hz sawtooth but it's similar to what you'd hear. Then you'll hear the same samples played back through a comb filter. Study the code for details.

Reverb

You can simulate reverb with a multi-tap delay line. Play the CelloSolo.wav file and then run the exponentialDecayReverb.ck in the download folder to file to hear a simulated multi-tap delay line reverb.

Convolution Reverb

Say you have a recording of you playing an instrument at home. Then you walk into the concert hall and pop a balloon or clap your hands and make a recording of the reverberations of that noise in the concert hall. Then you take the FFT of your home recording and the FFT of the concert hall impulse response and multiply the two spectra together. You then take the InverseFFT of the resulting specturm. When you play back the real valued results you'll hear yourself playing in the concert hall.

Open Octave and cd to the convolutionReverb folder inside the download folder and run convolutionReverb.m.

END OF MUSIC 208 LABS