INSTITUTE OF SONOLOGY

Royal Conservatoire The Hague

Modulation

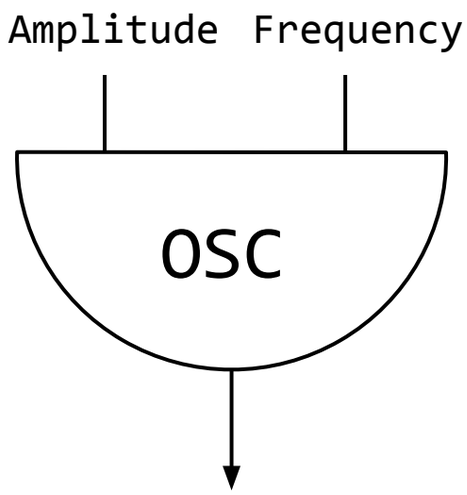
Programming and Music
http://www.bjarni-gunnarsson.net

Oscillators

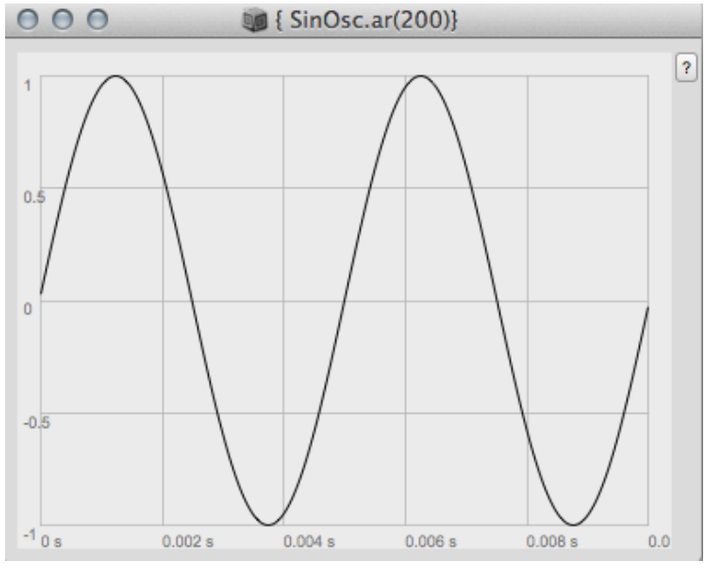
An ***oscillator*** produces an audio signal (waveform) at a specified ***frequency*** and ***amplitude***.

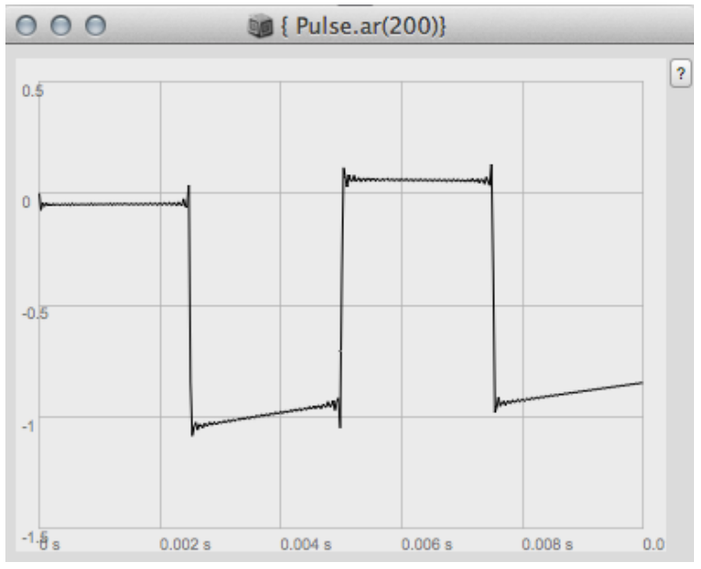
Digital implementations usually store one period of an output waveform in a ***table*** and then periodically read this table.

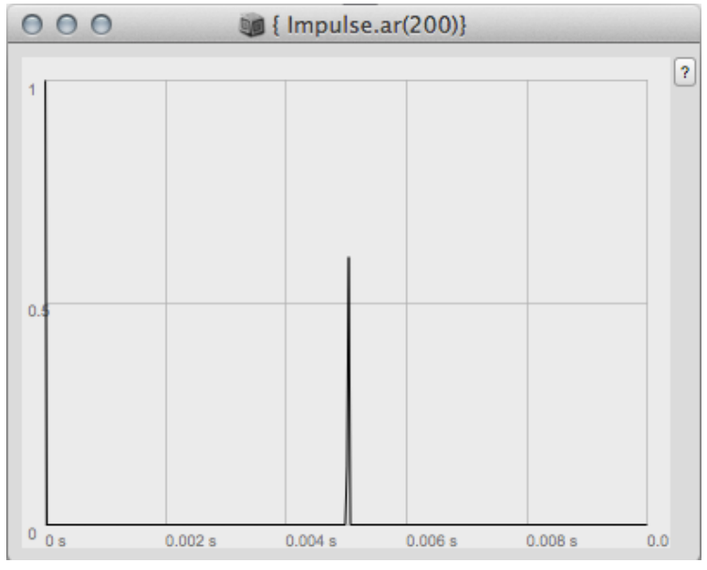
Common waveforms include ***sine, sawtooth, square*** and ***triangle*** waves.

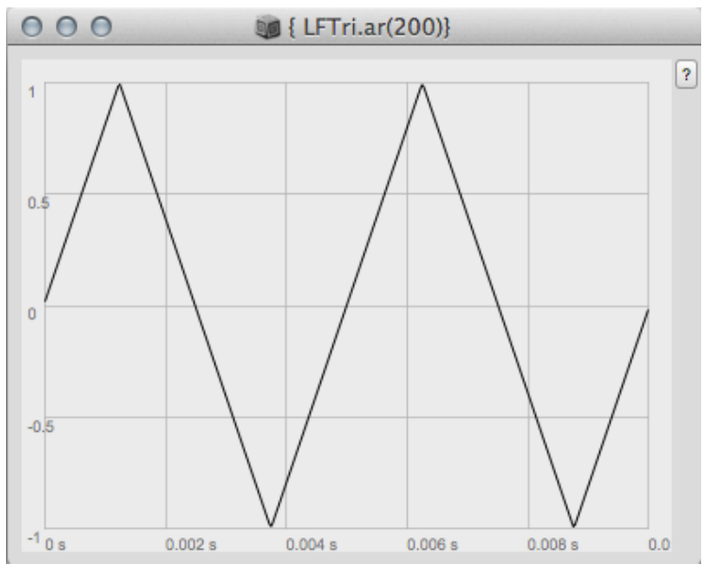


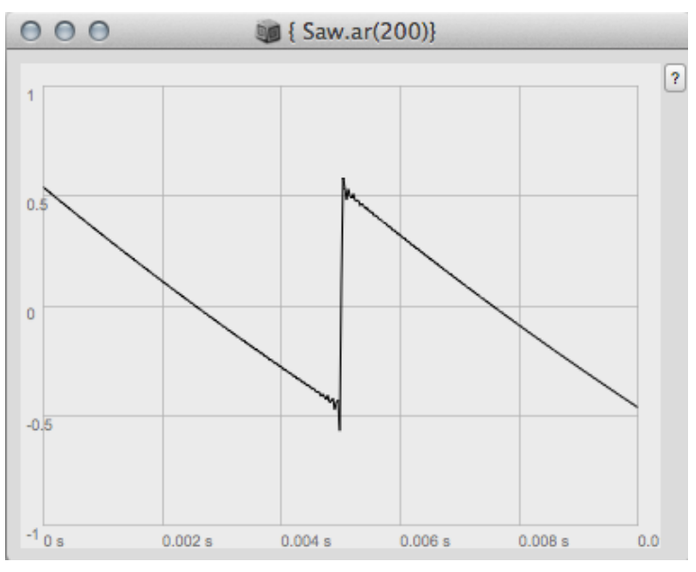
Oscillators

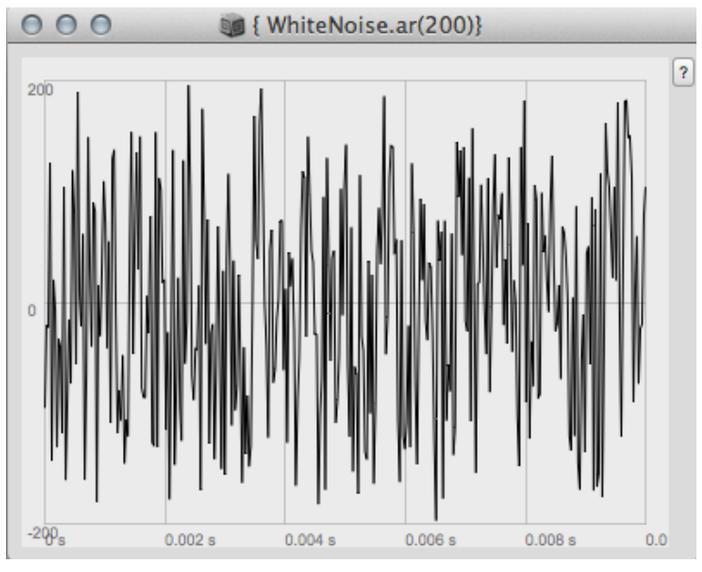










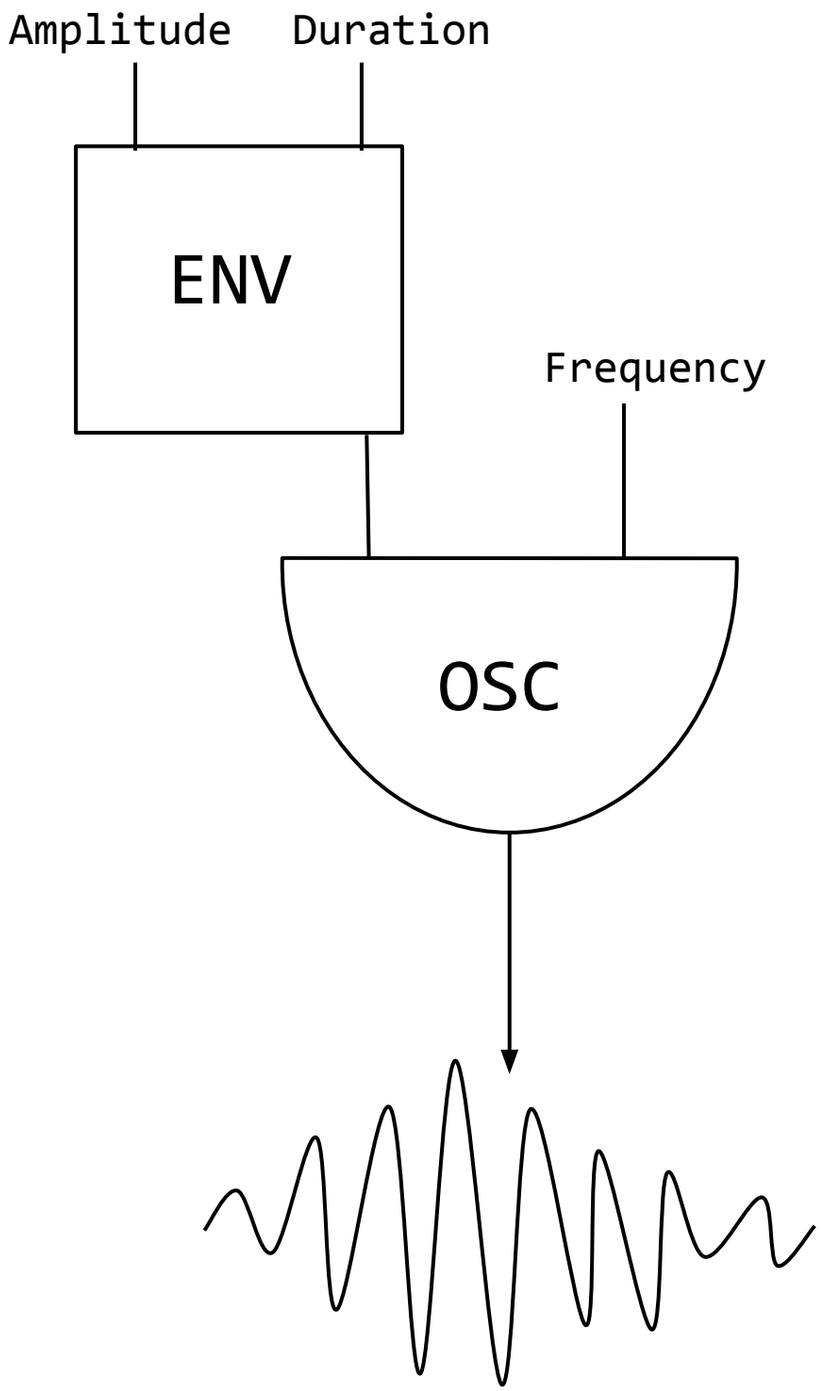


Envelopes

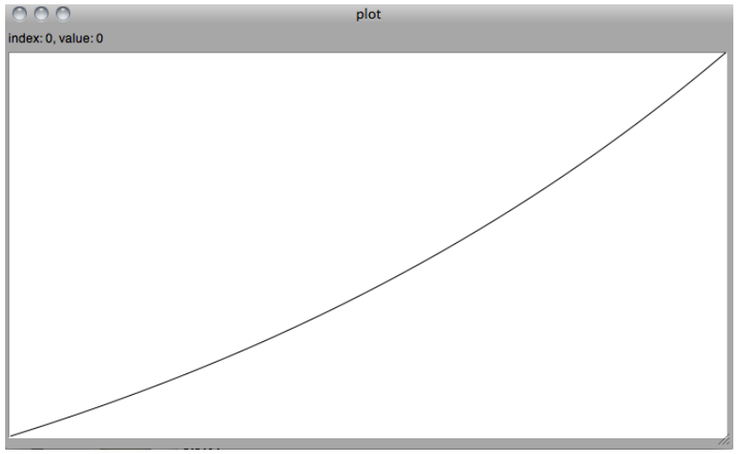
An ***envelope generator*** produces dynamically varying output over a specified duration. It is commonly used to vary the amplitude of an oscillator but also for controlling filters and many other synthesis parameters.

The main controls specify how the signal varies. An ***ADSR*** envelope has controls for ***attack, decay, sustain*** and ***release*** of the envelope

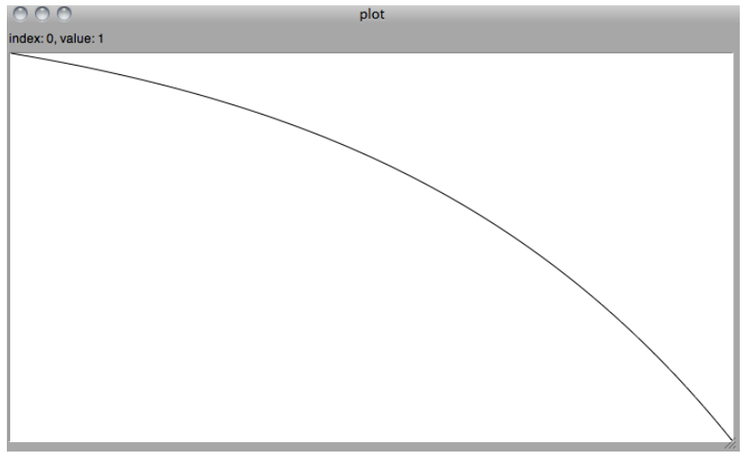
Envelopes



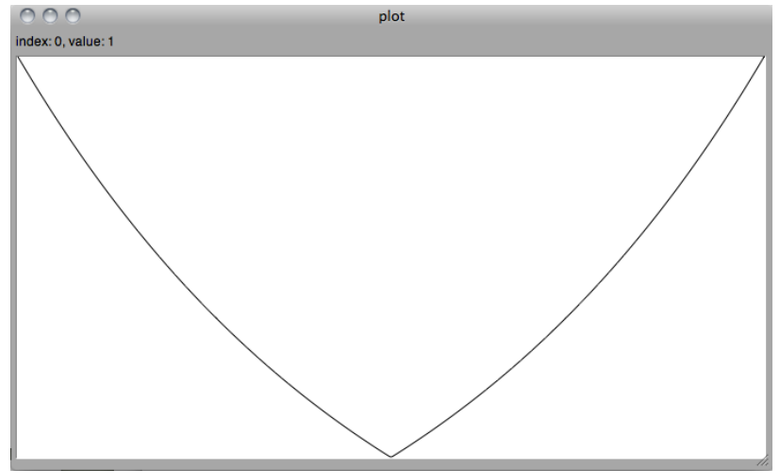
Envelopes



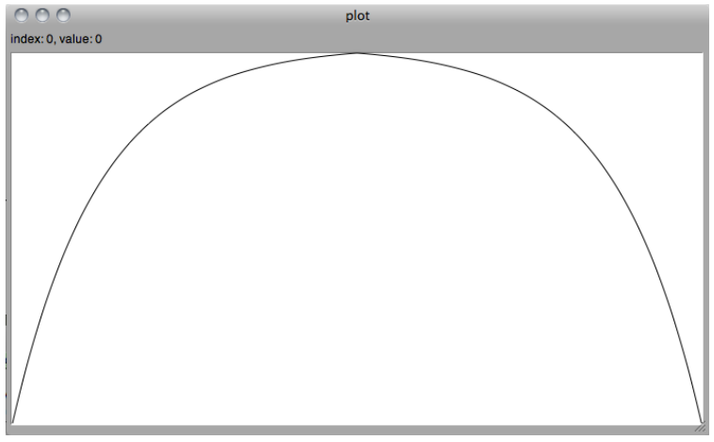
Ascent



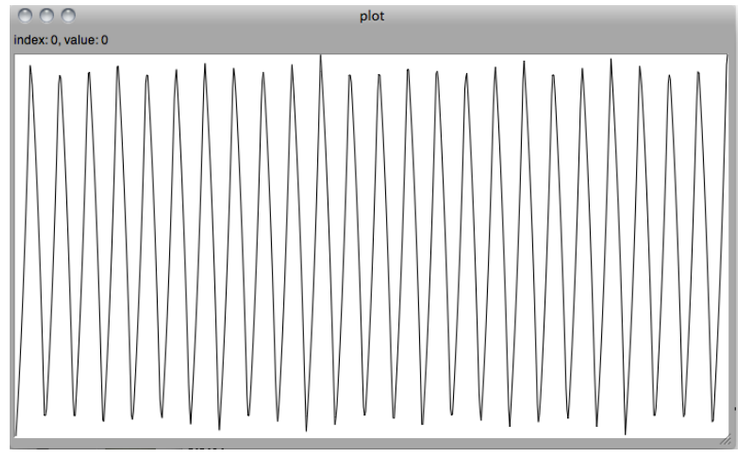
Descent



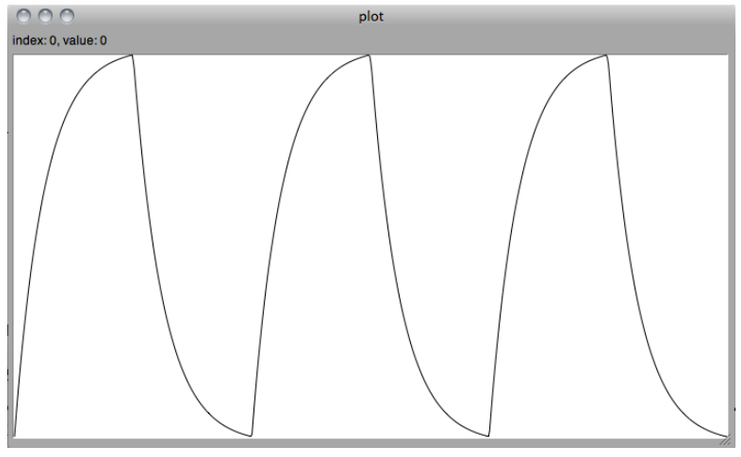
Parabola



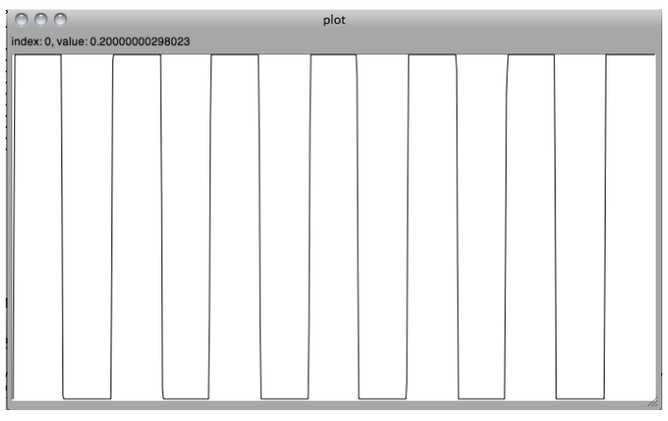
Inverse parabola



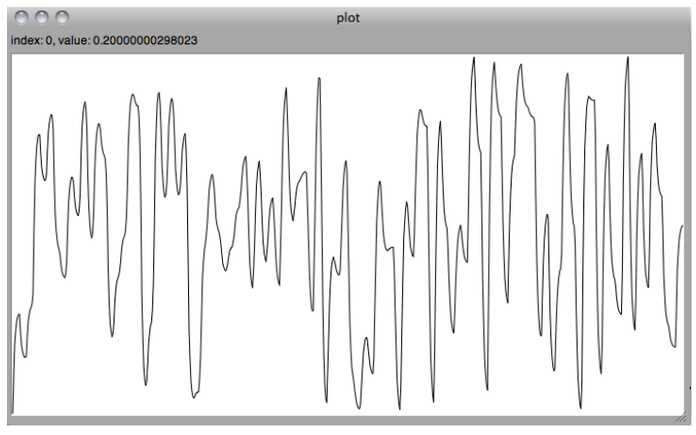
Oscillating



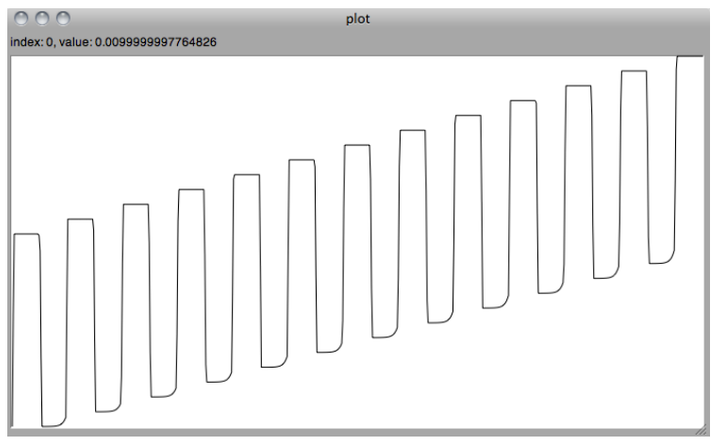
Undulation



Square

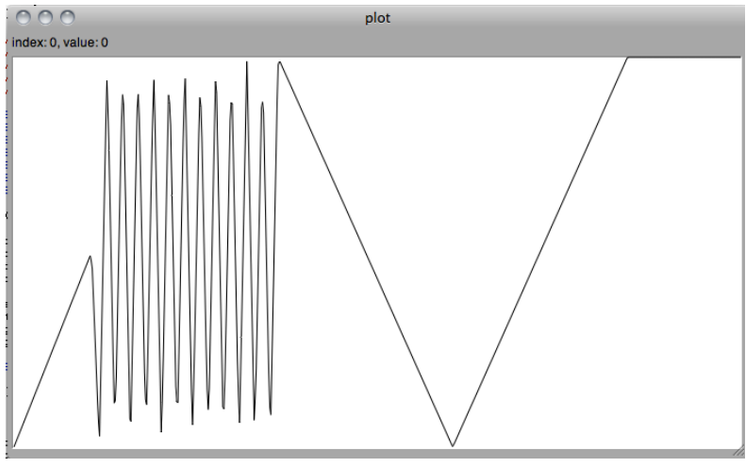


Random

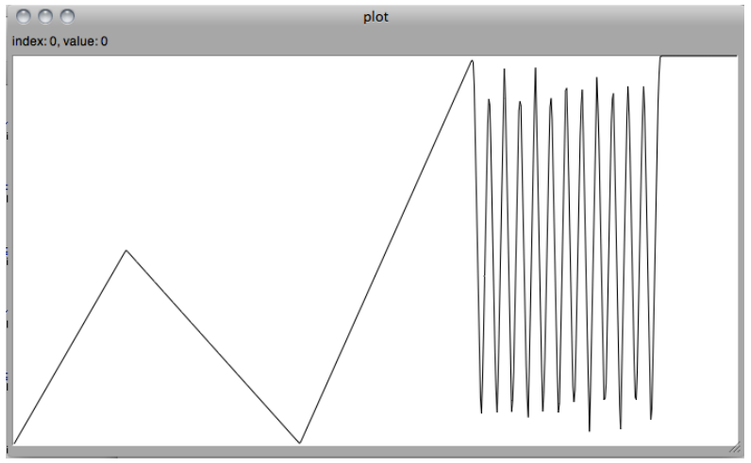


Steps

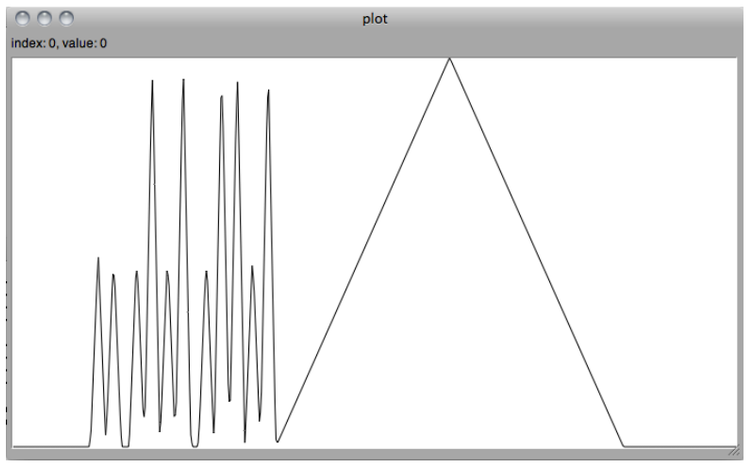
Envelopes



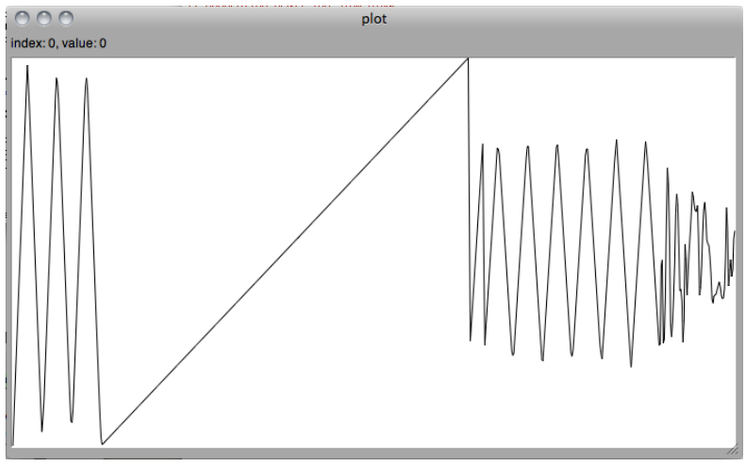
Composed 1



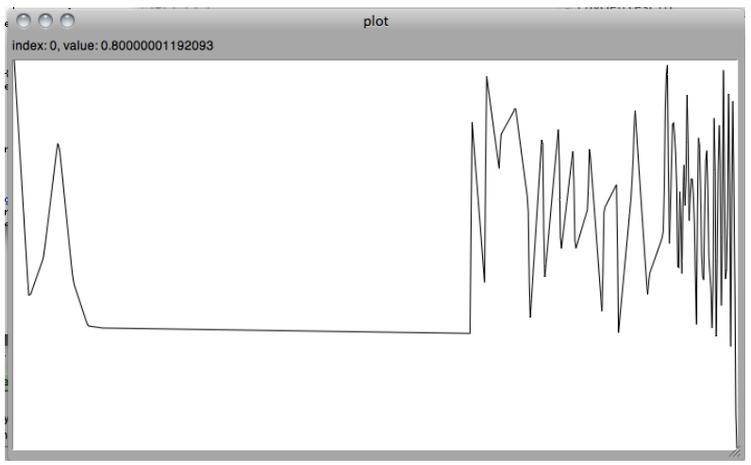
Reverse times



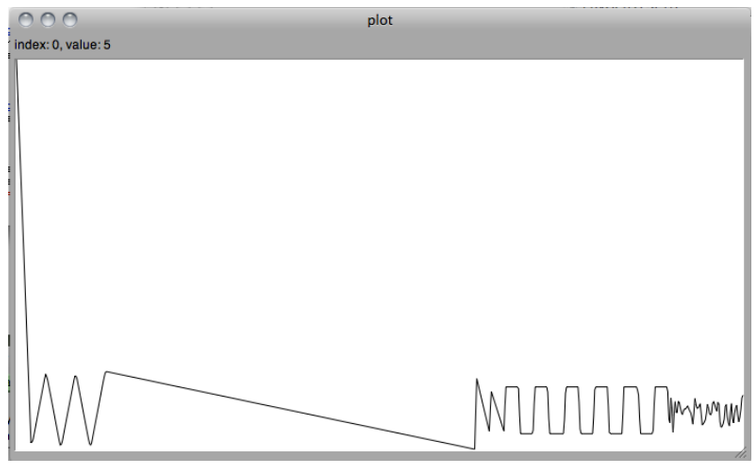
Pyramid



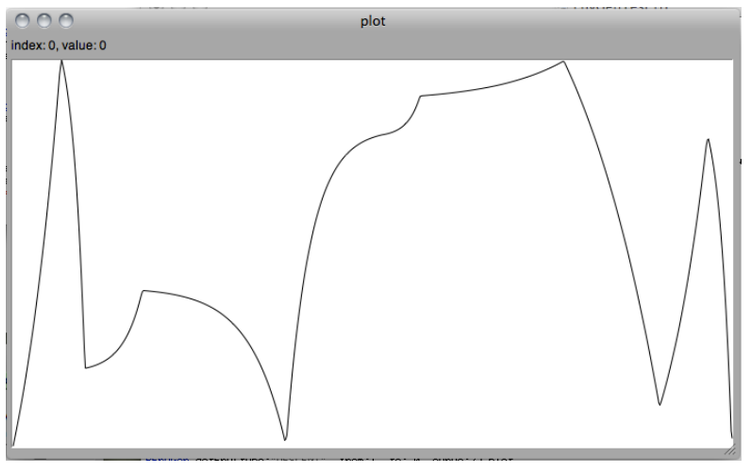
Composed 2



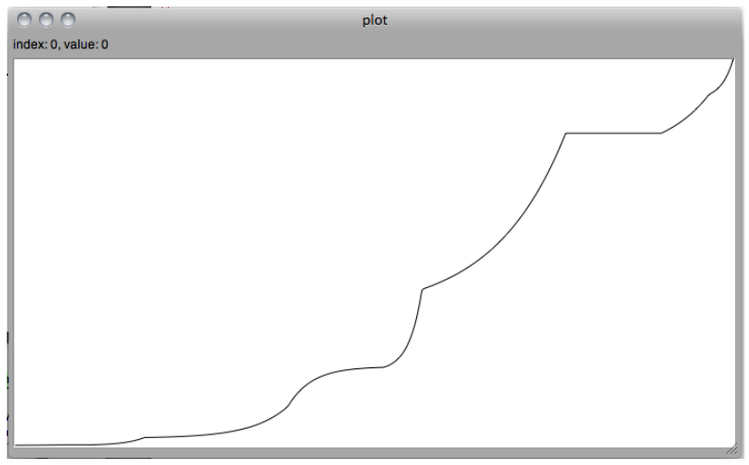
Reverse levels



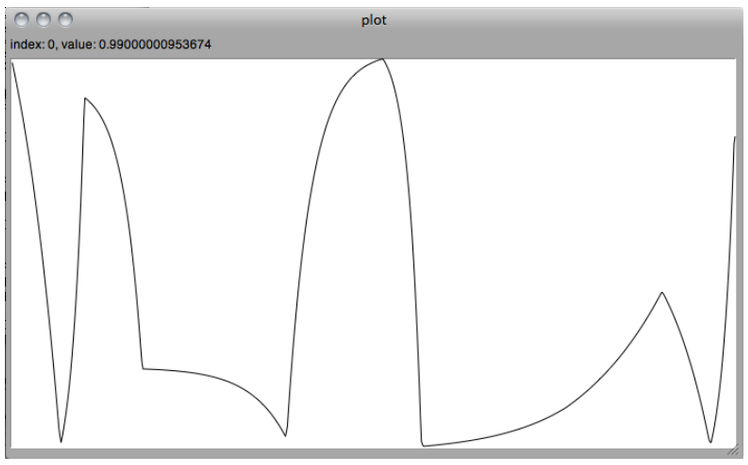
Shift by 5



Handmade



Sort levels



Scramble levels

SynthDefs

SynthDef = Synthesis Definition.

The normal workflow in SuperCollider is create a ***SynthDef***, add it to the server, then sequence musical events by creating new synths based on the synthdef and changing the arguments of these synths.

The SuperCollider ***server*** has to be booted before SynthDefs can be added.

SynthDefs and audio code on the server is fixed once the SynthDef has been added. This means ***loops*** or ***dynamic arrays*** can not be used in SynthDefs.

SynthDefs

The standard way of generating sounds in SuperCollider is by writing ***SynthDefs*** that result in a ***synthesis graph*** being created on the SuperCollider Server.

A synth definition is data about ***UGens*** and how they're connected. This is sent in a kind of special optimized form, called 'byte code', which the server can deal with very efficiently.

Once the server has a synth definition, it's can very efficiently use it to make a number of synths based on it. Synths on the server are basically just things that make or process sound, or produce control signals to drive other synths.

SynthDefs

The ***SynthDef class*** encapsulates the ***client-side representation*** of a given synthesis definitions. It also provides methods for creating new definitions, writing them to disk, and streaming them to a server.

It is important to understand that although a single def can provide a great deal of flexibility it is nevertheless ***a static entity***. A def's UGen graph function is evaluated only when the def is created. Statements like while, do, collect etc. will have no further effect at the time the def is used to create a Synth, and it is important to understand that a UGen graph function should not be designed in the same way as functions in the language, It will be evaluated ***once*** and only once.

Adding SynthDefs

In SuperCollider there are different ways of adding SynthDefs to the server.

The default one is ***.add*** and in most cases this one should be used. It sends the SynthDef to the server at the moment it is called and and does not create any additional file.This requires the program to initially to always add SynthDefs before it uses them.

For permanent existence on disk one could use either ***.writeDefFile*** or ***.store.***The former only writes the definition to disc while the second one also does, so but additionally adds it to the server. In most cases .store should be used for persistence storage.

SynthDefs

Arguments for the synth (inputs)

Name of synth

```
SynthDef(\mySynth, {
    |amp=0.1, freq=440, pan=0|
    var env, sig;
    env = EnvGen.kr(Env.perc, doneAction:2);
    sig = SinOsc.ar(freq) * env;
    sig = Pan2.ar(sig, pan, amp);
    Out.ar(0, sig);
}).add;
```

Variable for the synth (storage boxes)

Ugens that generate and modify signals

Final signal is sent to an output bus

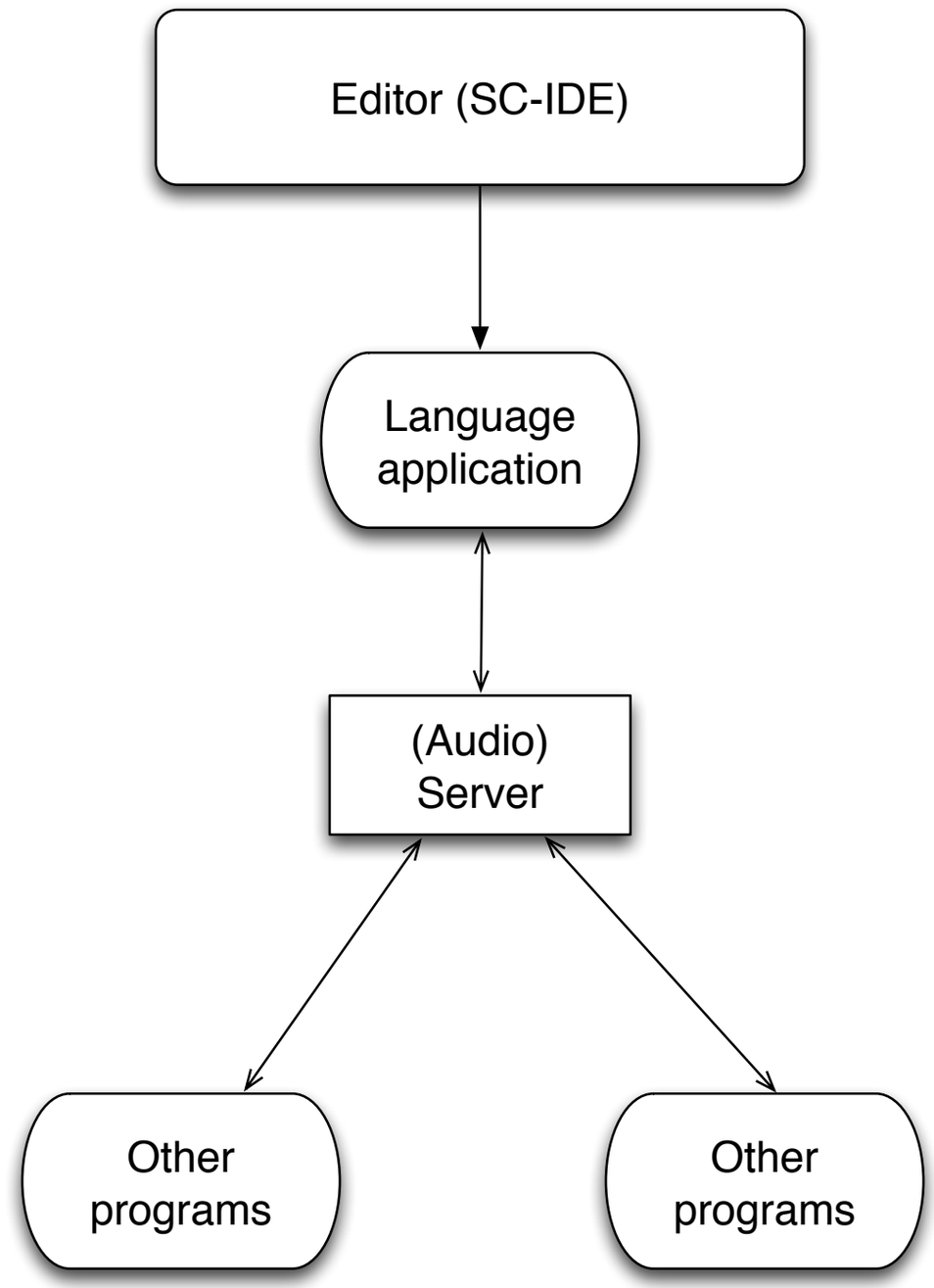
SynthDefs need to be added to the synthesis server

Architechture

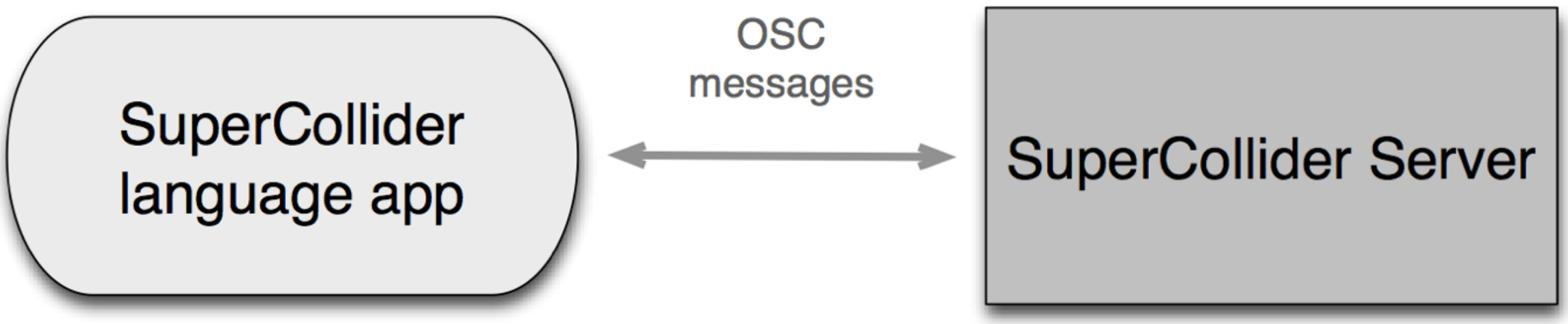
SuperCollider

supercollider.sourceforge.net

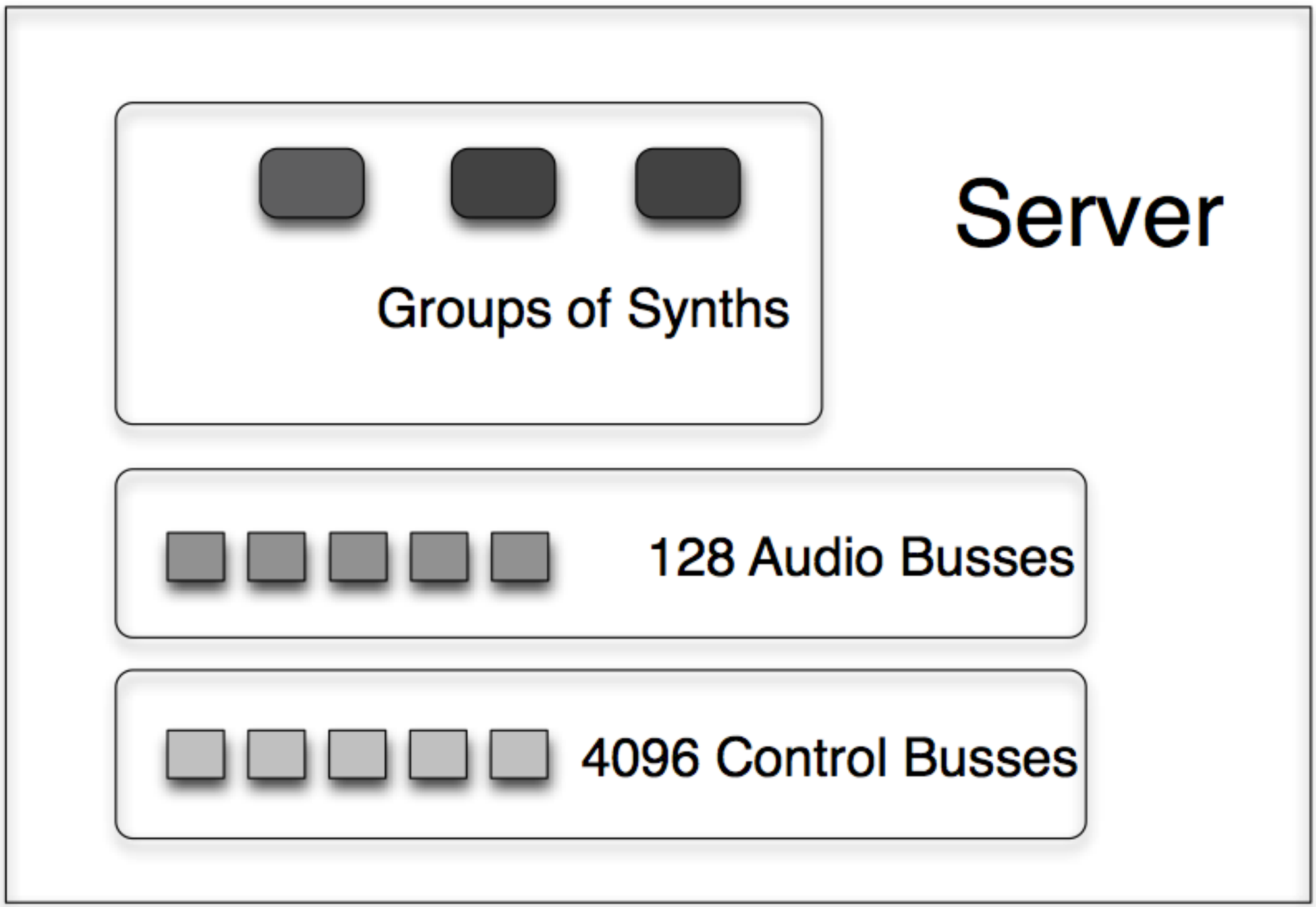
Version 3.6



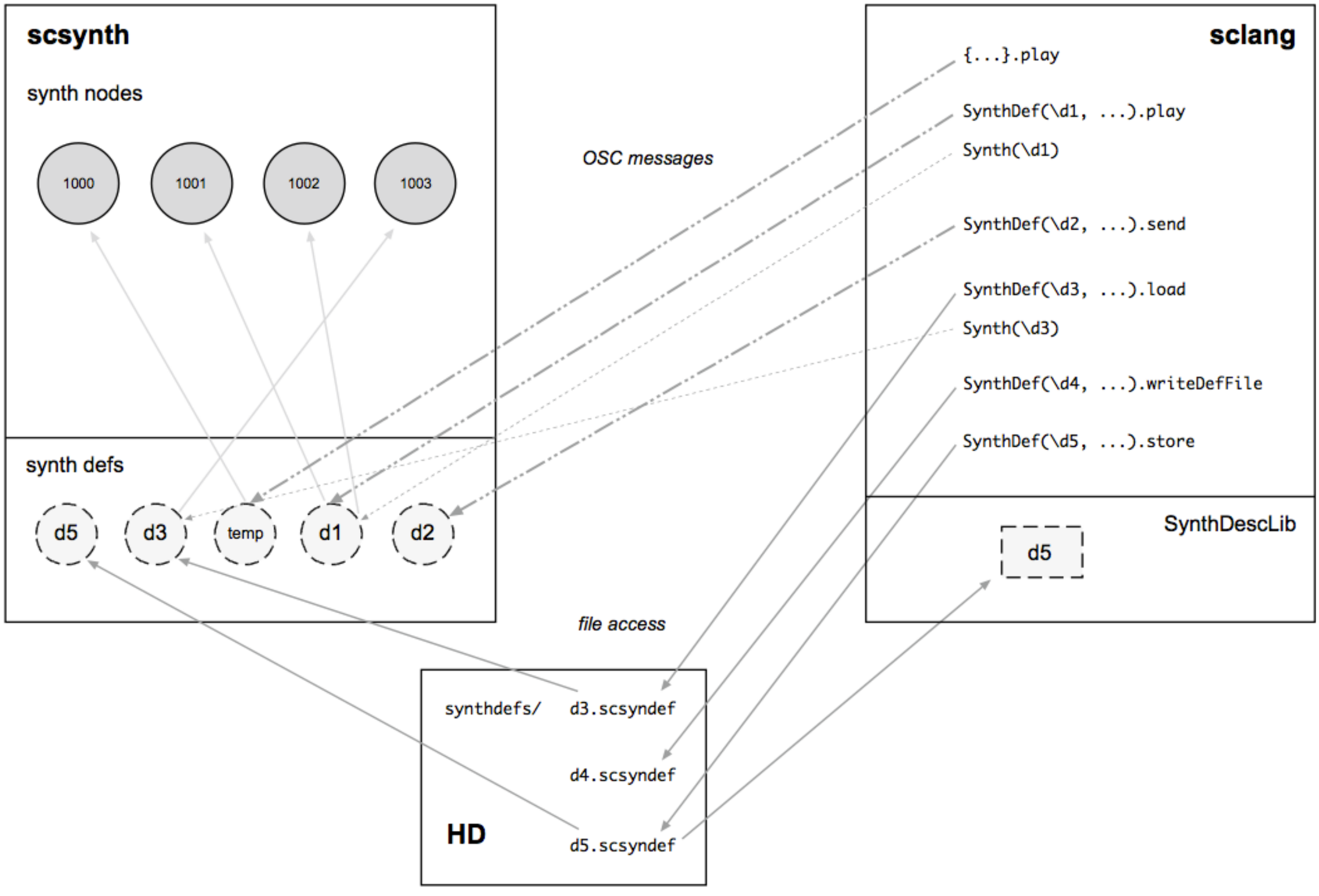
Architechture



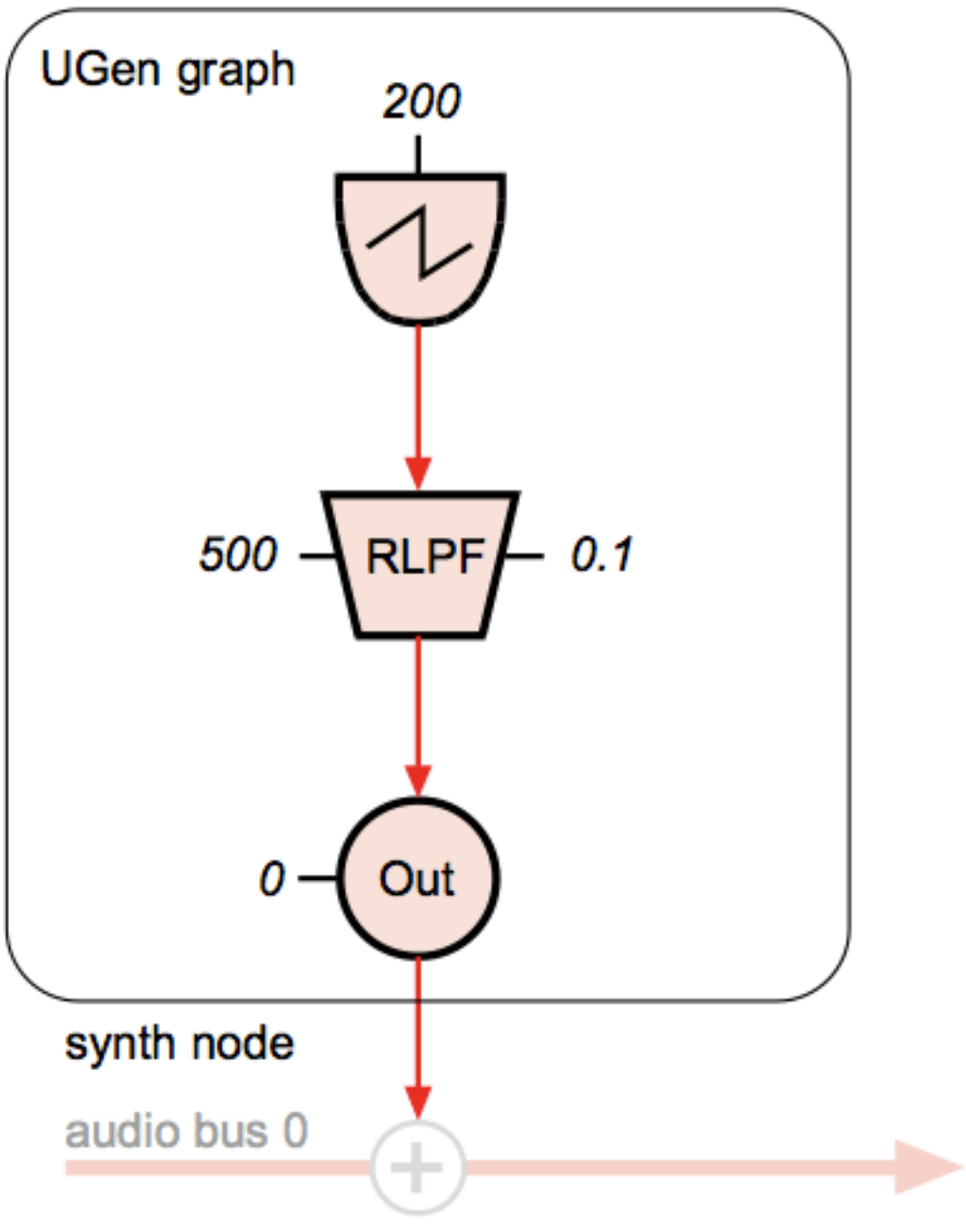
Architechture



Architechture



Architechture



Synth Nodes and UGen graphs

```
(
SynthDef("simple", {
var sig;
sig = Saw.ar(200);
sig = RLPF.ar(sig, 500, 0.1);
Out.ar(0, sig);
}).play;
)
```

Scaling

Unit generator come in slightly different ranges. Most generate signals between -1 and 1. These are called ***Bipolar*** Ugens while others generate signal between 0 and 1. These are called ***Unipolar*** Ugens.

Many Ugens contain special arguments that are used to scale their output, ***Mul*** and Add. ***Mul*** specifies a value used to multiply the signal and Add a value that will be added to the signal.

For Bipolar Ugens it easy to think of add as the ***center value*** of the range and mul the ***deviation*** above and below the center. To obtain the lowest value we use add minus mul, the highest is add plus mul.

For Unipolar Ugens add represents the low end of the range, mul + add the high end and mul the deviation.

Scaling

The ***range*** method can also be used to scale the output of a Ugen where is is enough to specify the low and hi limits of the destination range.

Mapping from one range to another can be done using various methods such as ***linlin, linexp, explin, expexp***.

Rates

There are two rates of calculation in SuperCollider, ***audio rate*** and ***control rate***. Audio rate is indicated with .ar and control rate with .kr.

The amount of numbers calculated per second with .kr is much lower, with a default sample rate .ar generates 44100 numbers per second, while .kr generates 700 values per second. The .kr amount is sample rate divided by control period (64).

The choice of whether to use audio or control rate depends on the application. Control rate uses less resources but some claim the slower rate causes stepwise motion and that things do not sound as well.

Multichannel expansion

Using arrays for single value arguments inside Ugens will cause the entire patch to be ***duplicated*** for each value of the array. Each resulting value will be routed to different channel meaning this is a very useful process for multichannel audio.

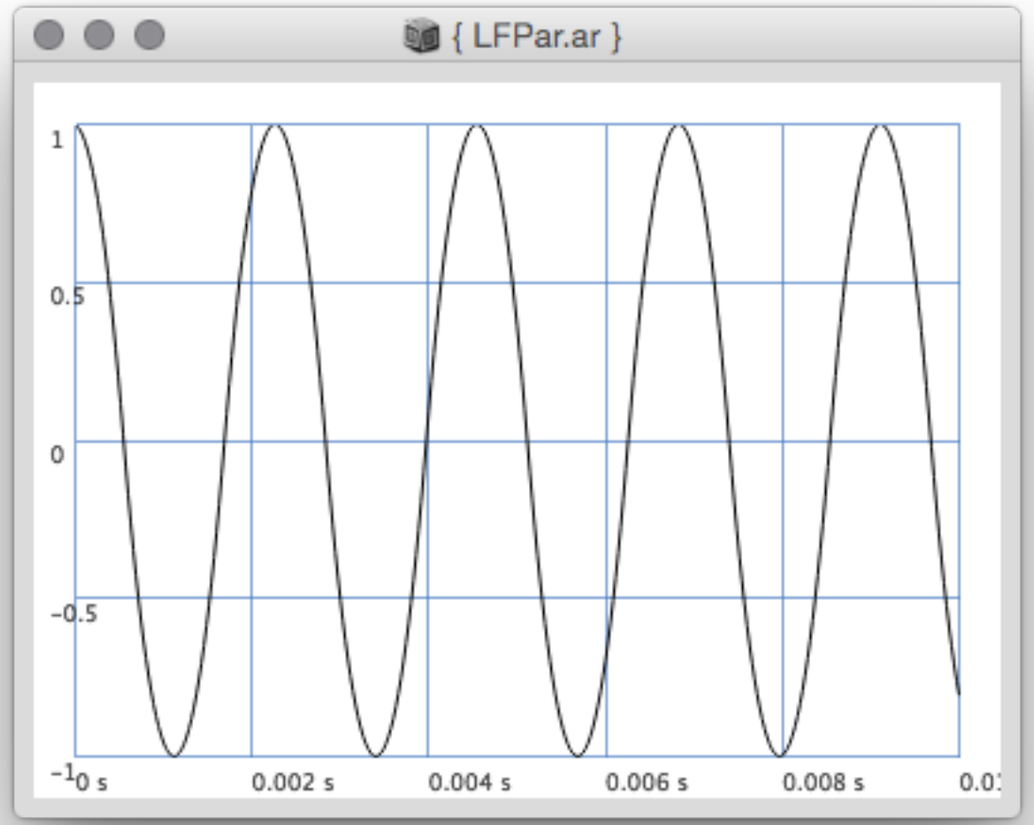
Using UGens such as ***Mix*** and ***Splay*** several channels can be mixed together so that multichannel expansion can also be used expressively for mono and stereo signals.

UGens

Oscillators

Sine wave variants:
SinOsc, FSinOsc, LFCub, LFPar, LFTri.

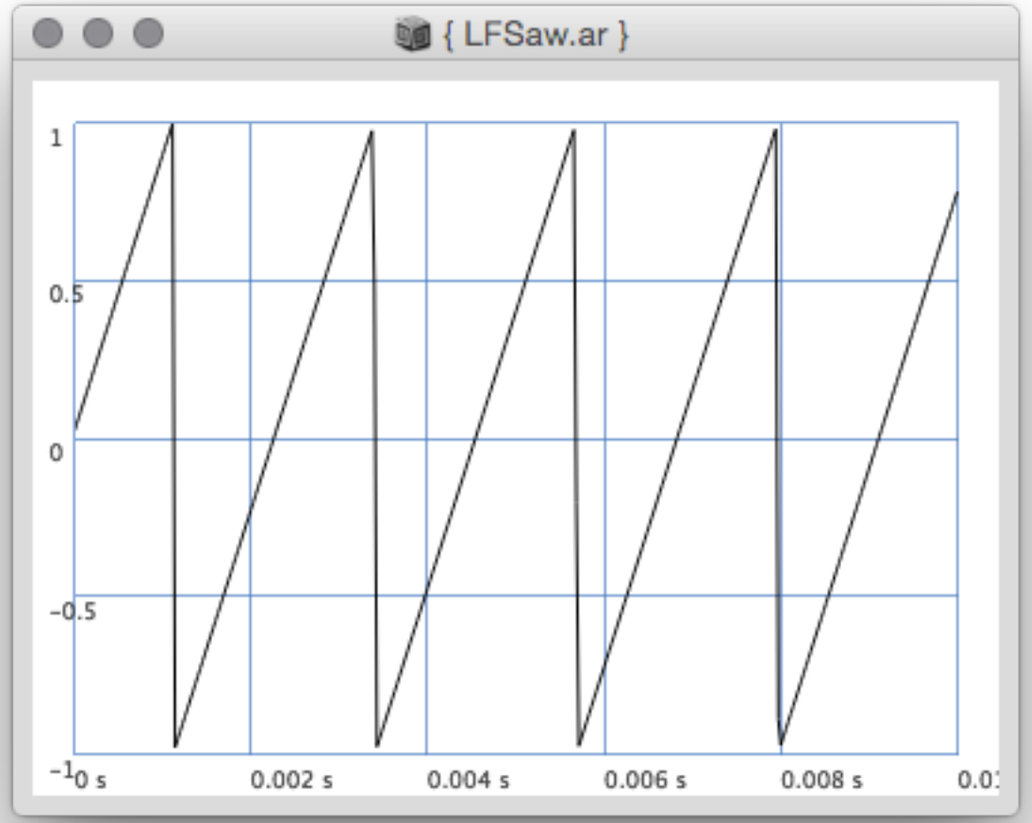
SinOsc.ar (freq: 440, phase: 0, mul: 1, add: 0) // Interpolating sine wavetable oscillator.
FSinOsc.ar (freq: 440, iphase: 0, mul: 1, add: 0) // Fast sine oscillator.
LFCub.ar (freq: 440, iphase: 0, mul: 1, add: 0) // A sine like shape made of two cubic pieces
LFPar.ar (freq: 440, iphase: 0, mul: 1, add: 0) // Parabolic oscillator
LFTri.ar (freq: 440, iphase: 0, mul: 1, add: 0) // Triangle oscillator

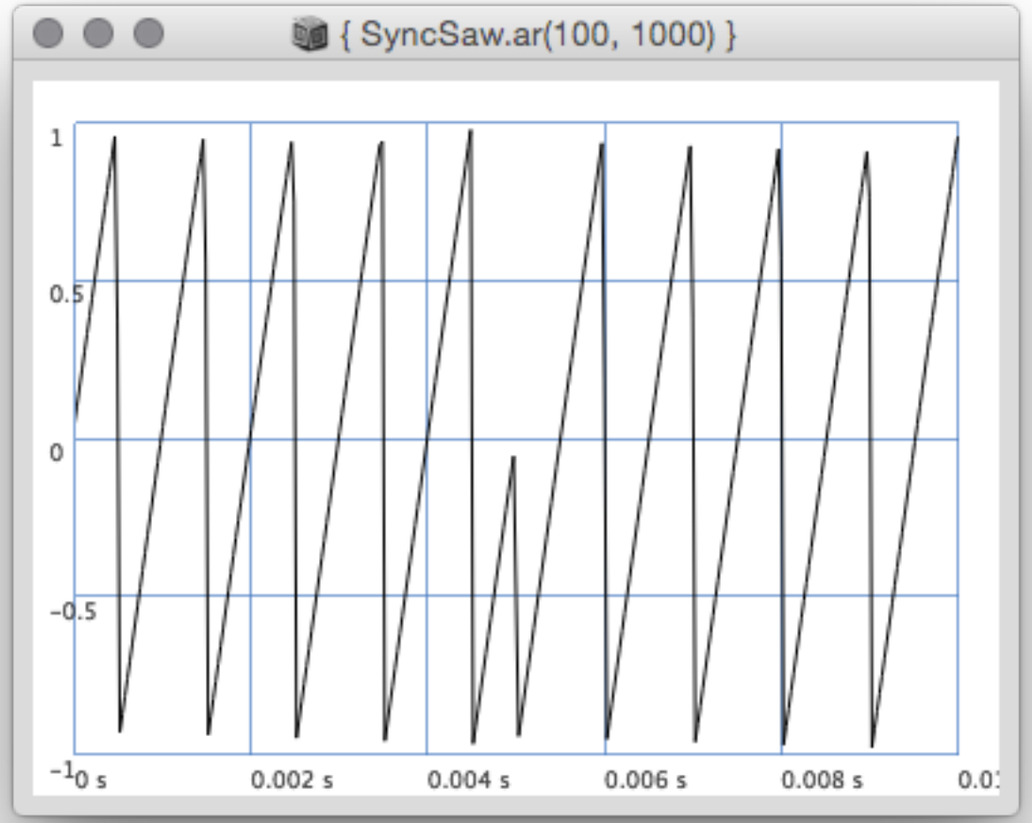


Oscillators

Saw wave variants:
Saw, VarSaw, SyncSaw.

Saw.ar (freq: 440, mul: 1, add: 0) // Band limited sawtooth
VarSaw.ar (freq: 440, iphase: 0, width: 0.5, mul: 1, add: 0) // Variable duty saw
SyncSaw.ar (syncFreq: 440, sawFreq: 440, mul: 1, add: 0) // Hard sync sawtooth wave.

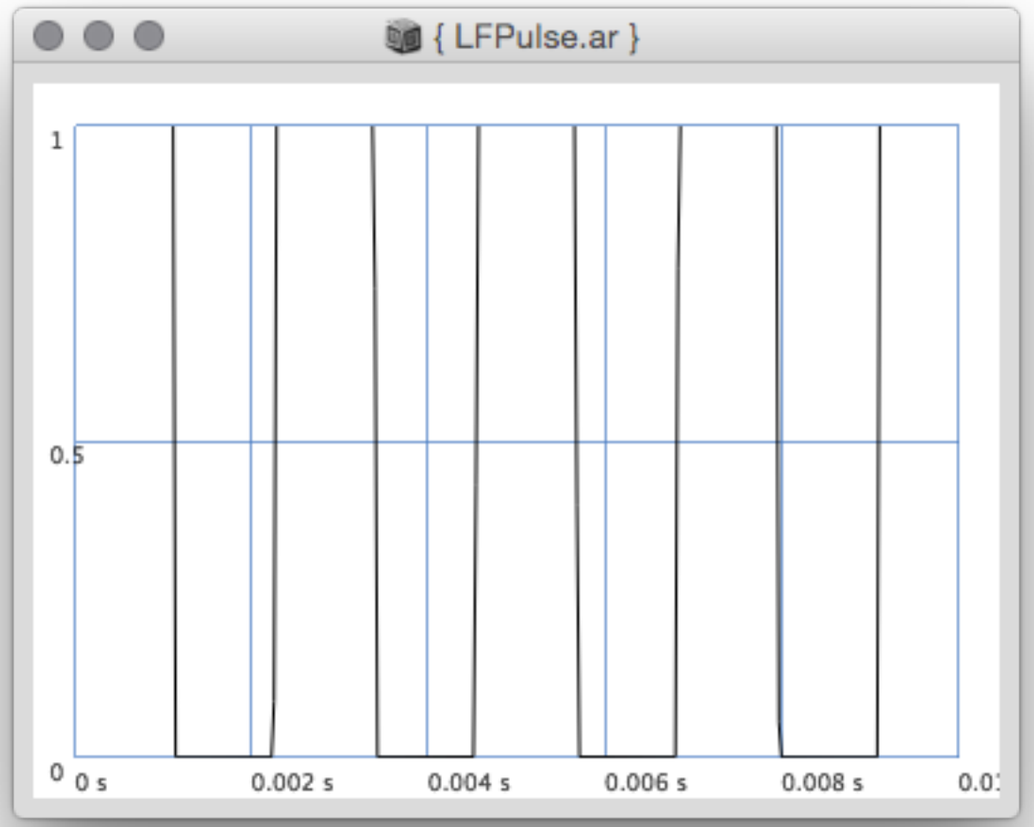


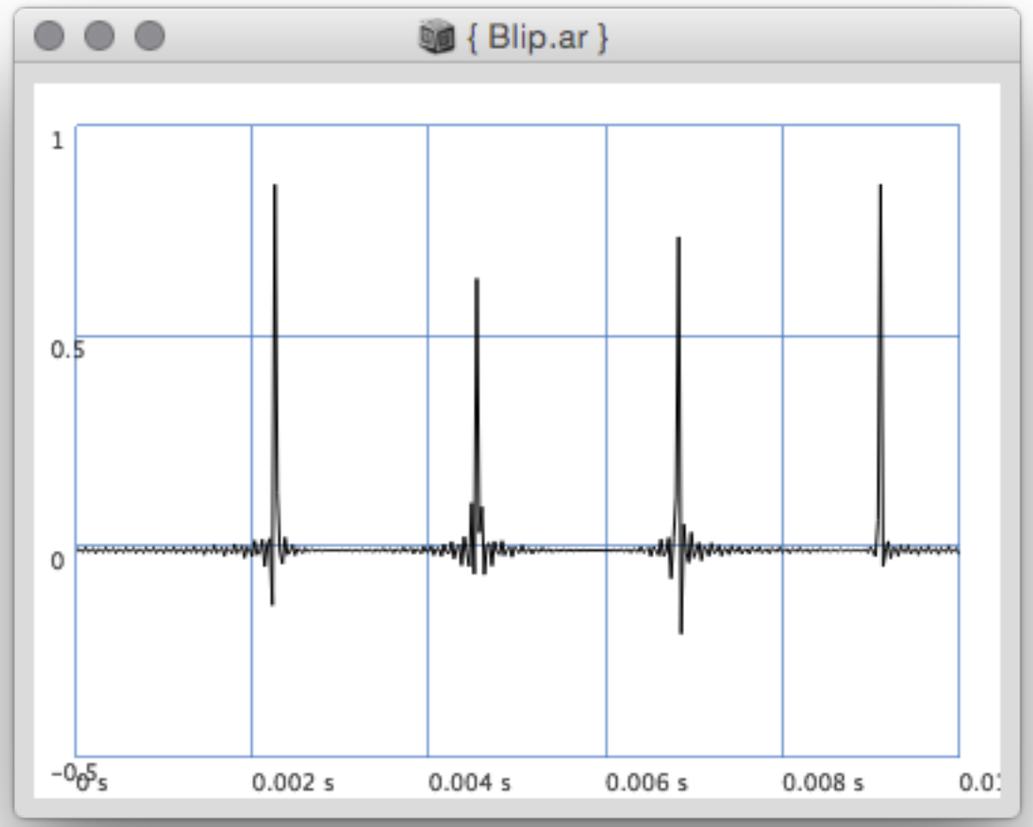


Oscillators

Pulse/Impulse wave variants:
Pulse, LFPulse. Impulse, Blip, Dust.

LFPulse.ar (freq: 440, iphase: 0, width: 0.5, mul: 1, add: 0) // Pulse oscillator
Pulse.ar (freq: 440, width: 0.5, mul: 1, add: 0) // Band limited pulse wave
Impulse.ar (freq: 440, phase: 0, mul: 1, add: 0) // Impulse oscillator
Dust.ar (density: 0, mul: 1, add: 0) // Random impulses
Dust2.ar (density: 0, mul: 1, add: 0) // Random impulses (-1 to 1)
Blip.ar (freq: 440, numharm: 200, mul: 1, add: 0) // Band limited impulse oscillator





Filters

Low Pass, High Pass
LPF, HPF - 12 dB / octave
arguments: in, freq, mul, add
```
{ LPF.ar(WhiteNoise.ar, MouseX.kr(1e2,2e4,1), 0.2) }.scope(1);
{ HPF.ar(WhiteNoise.ar, MouseX.kr(1e2,2e4,1), 0.2) }.scope(1);
{ LPF.ar(Saw.ar(100), MouseX.kr(1e2,2e4,1), 0.2) }.scope(1);
{ HPF.ar(Saw.ar(100), MouseX.kr(1e2,2e4,1), 0.2) }.scope(1);
```

Band Pass, Band Cut
BPF, BRF - 12 dB / octave
arguments: in, freq, rq, mul, add
rq is the reciprocal of the Q of the filter, or in other words: the bandwidth in Hertz = rq * freq.
```
{ BPF.ar(WhiteNoise.ar, MouseX.kr(1e2,2e4,1), 0.4, 0.4) }.scope(1);
{ BRF.ar(WhiteNoise.ar, MouseX.kr(1e2,2e4,1), 0.4, 0.2) }.scope(1);
{ BPF.ar(Saw.ar(100), MouseX.kr(1e2,2e4,1), 0.4, 0.4) }.scope(1);
{ BRF.ar(Saw.ar(100), MouseX.kr(1e2,2e4,1), 0.4, 0.2) }.scope(1);
```

Resonant Low Pass, High Pass, Band Pass
RLPF, RHPF - 12 dB / octave
arguments: in, freq, rq, mul, add
```
{ RLPF.ar(WhiteNoise.ar, MouseX.kr(1e2,2e4,1), 0.2, 0.2) }.scope(1);
{ RHPF.ar(WhiteNoise.ar, MouseX.kr(1e2,2e4,1), 0.2, 0.2) }.scope(1);
{ RLPF.ar(Saw.ar(100), MouseX.kr(1e2,2e4,1), 0.2, 0.2) }.scope(1);
{ RHPF.ar(Saw.ar(100), MouseX.kr(1e2,2e4,1), 0.2, 0.2) }.scope(1);
```

Resonz - resonant band pass filter with uniform amplitude
arguments: in, freq, rq, mul, add
```
{ Resonz.ar(WhiteNoise.ar(0.5), XLine.kr(1000,8000,10), 0.05) }.scope(1);
{ Resonz.ar(WhiteNoise.ar(0.5), 2000, XLine.kr(1, 0.001, 8)) }.scope(1);
{ Resonz.ar(WhiteNoise.ar(0.5), 2000, XLine.kr(0.001, 1, 8)) }.scope(1);
```

Filters

Ringz - ringing filter.
arguments: in, frequency, ring time, mul, add
Internally it is the same as Resonz but the bandwidth is expressed as a ring time.

```
{ Ringz.ar(Dust.ar(3, 0.3), 2000, 2) }.scope(1, zoom:4);
{ Ringz.ar(WhiteNoise.ar(0.005), XLine.kr(100,3000,10), 0.5) }.scope(1, zoom:4);
```

Simpler Filters
OnePole, OneZero - 6 dB / octave

```
{ OnePole.ar(WhiteNoise.ar(0.5), MouseX.kr(-0.99, 0.99)) }.scope(1);
{ OneZero.ar(WhiteNoise.ar(0.5), MouseX.kr(-0.49, 0.49)) }.scope(1);
```

NonLinear Filters
Median, Slew

```
// a signal with impulse noise.
{ Saw.ar(500, 0.1) + Dust2.ar(100, 0.9) }.scope(1);
// after applying median filter
{ Median.ar(3, Saw.ar(500, 0.1) + Dust2.ar(100, 0.9)) }.scope(1);
// a signal with impulse noise.
{ Saw.ar(500, 0.1) + Dust2.ar(100, 0.9) }.scope(1);
// after applying slew rate limiter
{ Slew.ar(Saw.ar(500, 0.1) + Dust2.ar(100, 0.9),1000,1000) }.scope(1);
```

Formant Filter
Formlet - A filter whose impulse response is similar to a FOF grain.

```
{ Formlet.ar(Impulse.ar(MouseX.kr(2,300,1), 0, 0.4), 800, 0.01, 0.1) }.scope(1, zoom:4);
```

Klank - resonant filter bank
arguments: `[frequencies, amplitudes, ring times], mul, add

```
{ Klank.ar(`[[200, 671, 1153, 1723], nil, [1, 1, 1, 1]], Impulse.ar(2, 0, 0.1)) }.play;
{ Klank.ar(`[[200, 671, 1153, 1723], nil, [1, 1, 1, 1]], Dust.ar(8, 0.1)) }.play;
{ Klank.ar(`[[200, 671, 1153, 1723], nil, [1, 1, 1, 1]], PinkNoise.ar(0.007)) }.play;
```

Exercises

```
(
NF(\iop, {|freq=78, mul=1.0, add=0.0|
    var noise = LFNoise1.ar(0.001).range(freq, freq + (freq * 0.1));
    var osc = SinOsc.ar([noise, noise * 1.04, noise * 1.02, noise * 1.08],0,0.2);
    var out = DFM1.ar(osc,freq*4,SinOsc.kr(0.01).range(0.92,1.05),1,0,0.005,0.7);
    HPF.ar(out, 40)
}).play;
)

(
NF(\dsc, {|freq = 1080|
    HPF.ar(
        BBandStop.ar(Saw.ar(LFNoise1.ar([19,12]).range(freq,freq*2), 0.2).excess(
        SinOsc.ar( [freq + 6, freq + 4, freq + 2, freq + 8])),
        LFNoise1.ar([12,14,10]).range(100,900),
        SinOsc.ar(20).range(9,11)
    ), 80)
    ;
}).play;
)
```

```
var <>pindex, <>cindex;

initialize {
    if(pindex.isNil, { pindex = 1000 });
    if(cindex.isNil, { cindex = 2000 });
}

clearProcessSlots {
    pindex = 1000;
    (this.pindex - 1000).do{|i| this[this.pindex+i] = nil; }
}

clearOrInit {|clear=true|
    if(clear == true, { this.clearProcessSlots() }, { this.initialize() });
}

transform {|process, index|
    if(index.isNil && pindex.isNil, {
        this.initialize();
    });

    pindex = pindex + 1;
    this[pindex] = \filter -> process;
}

control {|process, index|
    var i = index;

    if(i.isNil, {
        this.initialize();
        cindex = cindex + 1;
        i = cindex;
    });

    this[i] = \pset -> process;
}
```

```
(
NF(\depfm, {|freqMin=5, freqMax=20, mul=20, add=80, rate=0.5, modFreq=2100, index=0.3, amp=0.2|
    var trig, seq, freq;
    trig = Dust.kr(rate);
    seq = Diwhite(freqMin, freqMax, inf).midicps;
    freq = Demand.kr(trig, 0, seq);
    HPF.ar(PMOsc.ar(LFCub.kr([freq, freq/2, freq/3, freq/4], 0, mul, add),
    LFNoise1.ar(0.3).range(modFreq,modFreq*2), index) * amp, 50)
}).play;
)
```

Exercises

Exercise 1: Tremolo with Variable Rate

Create a tremolo effect on a sine wave where the tremolo rate itself changes slowly over time (use an LFO to control the tremolo rate).

Exercise 2: Amplitude Modulation Exploration

Create an AM synthesis patch where both the carrier frequency (300-600 Hz) and modulator frequency (50-300 Hz) are controlled by separate LFNoise generators. Use freqscope to observe the sidebands.

Exercise 3: Ring Modulation with Envelopes

Create a ring modulated sound that uses envelopes on both the carrier and modulator frequencies. The sound should have a clear attack and decay lasting about 8 seconds.

Exercise 4: FM Bell Sound

Create an FM bell-like sound using a carrier frequency of 440 Hz and a modulator ratio of 1.4. The modulation index should start at 10 and decay to 0 over 3 seconds. Add an amplitude envelope as well.