Remember: If you are going to be sending commands to the Server, you need to execute code like that following that assigns the current Server to the variable 's'.

```
(
    Server.default = Server.internal;
    s = Server.default;
    s.boot;
)

FreqScope.new(300, 200);
```

Unit Generators

A unit generator is an object that processes or generates sound. There are many classes for unit generators, all of which inherit from the class **UGen**.

Unit generators in SuperCollider can have many inputs, but always have a single output. Unit generator classes which would naturally have several outputs such as a panner, produce an array of output signals.

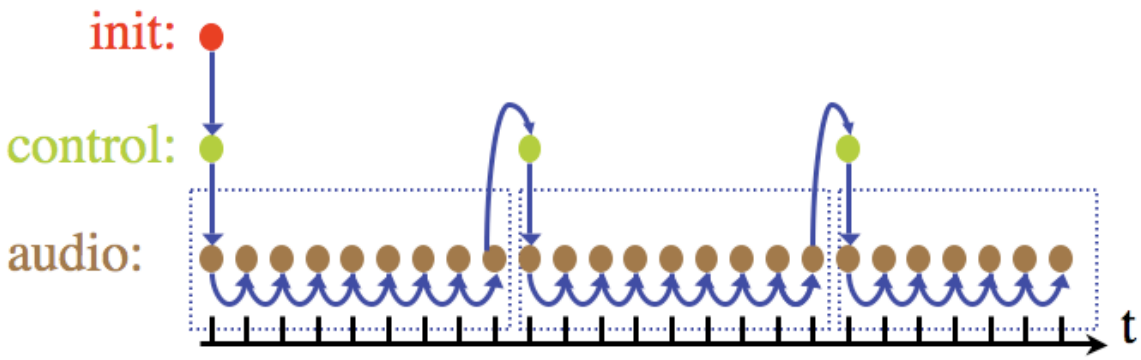
The input parameters for a unit generator are given in the documentation for that class.

```
SinOsc.ar(800, 0.0, 0.2); // create a sine oscillator at 800 Hz, phase 0.0, amplitude 0.2
```

Instantiation, Audio Rate, Control Rate

A unit generator is instantiated by sending the 'ar' or 'kr' message to the unit generator's class (instead of a 'new' message). Like nearly all audio programs, the signal flow runs at multiple rates. Some parts of the unit generator code that is interpreted once, because nothing changes after that. The 'kr' message instantiates a unit generator that runs at the control rate. The 'ar' message instantiates a unit generator that runs at audio rate.

Control rate unit generators are used for low frequency or slowly changing control signals and produce only a single sample per control cycle. They therefore use less processing power than audio rate unit generators. The default control rate is 689 Hz.



Arguments

A unit generator's arguments can be other unit generators, scalars, or arrays of unit generators and scalars.

One of the conventions of most Unit Generators is that their last two arguments are

mul and add

The default values are 1.0 and 0.0 respectively. Using the mul and add arguments is more efficient than writing expressions with multiplications and additions.

Keyword Arguments

Keyword arguments can be very helpful, especially for unit generators with lots of arguments.

Examples of using keyword arguments:

```
{ SinOsc.ar(801, 0, 0.1, 0) }.play;
```

freq, phase, mul, and add are the names of arguments

```
{SinOsc.ar(freq: 440, phase: 0, mul: 0.4, add: 0)}.play;
```

```
{SinOsc.ar(phase: 0, freq: 440, add: 0, mul: 0.4)}.play;
```

```
{SinOsc.ar(freq: 440, mul: 0.4)}.play;
```

Function Notation

```
{ SinOsc.ar(440, 0, 0.2) }.play;
```

This is probably the simplest way to get audio in SuperCollider and is often used

for demonstration and testing purposes. It automatically wraps the Function's code in a **SynthDef** (adding an **Out** UGen if needed), creates and starts a new **Synth** with it, and returns the Synth object. (If we don't store the Function in a variable, it can't be reused.)

```
(
{ var ampOsc;      // Open Function and declare variable
      ampOsc = SinOsc.kr(0.5, 1.5pi, 0.5, 0.5);
               // Make a control rate SinOsc
               // and assign its output signal to variable
      SinOsc.ar(440, 0, ampOsc);  // Make an audio rate SinOsc
               // and use the variable to control amplitude
}.play;
)
```

Example: ading sine waves and plotting them:

```
({
    var f=800;
    a = SinOsc.ar(f,0,0.1);
    b = SinOsc.ar(2*f,0,0.1);
    c = SinOsc.ar(3*f,0,0.1);
    d = SinOsc.ar(4*f,0,0.1);
    e = SinOsc.ar(5*f,0,0.1);
    a+b+c+d+e;
    }.scope;
)
```

If you write the Unit-Generator code as a SynthDef, you must include an output unit generator.

Out write a signal to a bus

superclass: AbstractOut

***ar(bus, channelsArray)** - write a signal to an audio bus.

bus - the index of the bus to write out to. The lowest numbers are written to the audio hardware. The output bus numbers start at 0 and go up by integer for each additional channels. The stereo output channels are 0 and 1.
channelsArray - an Array of channels or single output to write out. You cannot change the size of this once a SynthDef has been built.

```
SynthDef("sines", {
    var f=800;
    a = SinOsc.ar(f,0,0.1);
```

```
    b = SinOsc.ar(2*f,0,0.1);
    c = SinOsc.ar(3*f,0,0.1);
    d = SinOsc.ar(4*f,0,0.1);
    e = SinOsc.ar(5*f,0,0.1);
    Out.ar(0, a+b+c+d+e);  // this Out is necessary
}).load(s);

Synth("sines").play;
s.scope(1);
```

Some Plotting Basics

How to start **plotting**:

```
    { BrownNoise.ar(0.1) }.play;
```

and then,

```
    s.scope(1);
```

Or, you can write:

```
    { SinOsc.ar(800, 0, 0.1) }.scope(1);
```

Controlling a plot:

```
    s.scope(numChannels:1, zoom:1); // default values
    s.scope(numChannels:2); // Stereo
    s.scope(numChannels:1,zoom:4) // zoom out
    s.scope(numChannels:1, zoom:0.25);  // zoom in
```

Don't over-react to plot by-products

```
    { SinOsc.ar(689, 0, 0.1) }.scope(1)
    { SinOsc.ar(900.3, 0, 0.1) }.scope(1)
```

More On Periodic Sources

Look at the Help pages for the following:

SinOsc	FSinOsc	Osc
Buffer	OscN	

These are also useful:

Impulse	Blip	Pulse	LFPulse
Saw	LFSaw	SyncSaw	
Klang	Formant		
VOsc	VOsc3		

Band Limited Signal Generators

SinOsc, FSinOsc, Blip, Saw, Pulse, Formant will not alias.

SinOsc, FSinOsc

arguments: frequency, phase, mul, add

```
{ SinOsc.ar(800,0,0.5) }.scope(1, zoom: 0.5);
```

Example with main frequency foldover
(the one thing that defeats the band-limited design):

```
{ SinOsc.ar(Line.kr(1,43000,6), 0, 0.1) }.scope(1);

{ FSinOsc.ar(800,0,0.5) }.scope(1, zoom: 0.5);
// FSinOsc should not have its frequency modulated.
// Since it is based on a filter at the edge of stability, it will blow up:
```

Saw

arguments: frequency, mul, add

```
{ Saw.ar(800, 0.5) }.scope(1, zoom:0.5);
{ Saw.ar(XLine.kr(200,20000,6),0.5) }.scope(1, zoom: 4);  // no aliasing
```

Blip

arguments: frequency, numHarmonics, mul, add

```
{ Blip.ar(800, 10, 0.5) }.scope(1, zoom:0.5);
{ Blip.ar(XLine.kr(200,20000,6),10,0.5) }.scope(1, zoom: 0.5); // no aliasing
// modulate number of harmonics
{ Blip.ar(400,Line.kr(1,30,20),0.2) }.scope(1, zoom: 0.5);
//  thru a resonant low pass filter
{ RLPF.ar(Blip.ar(400,30,0.5), XLine.kr(400,8000,10), 0.1) }.scope(1, zoom: 0.5);
```

Pulse

arguments: frequency, width, mul, add

```
{ Pulse.ar(800,0.3, 0.5) }.scope(1, zoom: 0.5);
{ Pulse.ar(XLine.kr(200,20000,6),0.3,0.5) }.scope(1, zoom: 0.5);
// modulate pulse width
```

```
{ Pulse.ar(400, Line.kr(0.01,0.9,8), 0.5) }.scope(1, zoom: 0.5);
```

Klang - sine oscillator bank

arguments: `[frequencies, amplitudes, phases], mul, add

```
{ Klang.ar(`[ [800, 1000, 1200],[0.3, 0.3, 0.3],[pi,pi,pi] ], 1, 0) * 0.4}.scope(1, zoom: 0.5);
// using a random function to generate eight random frequencies
{ Klang.ar(`[ {exprand(400, 2000)}.dup(8), nil, nil ], 1, 0) * 0.04 }.scope(1, zoom: 0.5);
{ Klang.ar(`[ {exprand(400, 15000)}.dup(20), nil, nil ], 1, 0) * 0.04 }.scope(1, zoom: 0.5);
```

Formant formant oscillator

arguments: kfundfreq, kformfreq, kwidthfreq, mul, add

Generates a set of harmonics around a formant frequency at a given fundamental frequency.
kfundfreq - fundamental frequency in Hertz.
kformfreq - formant frequency in Hertz.
kwidthfreq - pulse width frequency in Hertz. Controls the bandwidth of the formant. Widthfreq must be greater than or equal fundfreq.

```
// modulate fundamental frequency, formant freq stays constant
{ Formant.ar(XLine.kr(400,1000, 8), 2000, 800, 0.125) }.play

// modulate formant frequency, fundamental freq stays constant
{ Formant.ar(200, XLine.kr(400, 4000, 8), 800, 0.125) }.play

// modulate width frequency, other freqs stay constant
{ Formant.ar(400, 2000, XLine.kr(800, 8000, 8), 0.125) }.play
```

Table-Based Signal Generators

Osc, COsc, VOsc, VOsc3

Use a buffer allocated on the server.

```
(
    // allocate buffer with server, size, numChans, bufferNumber
    b = Buffer.alloc(s, 2048, 1, bufnum: 80);
    // fill buffer with array of amplitudes, 3 * true (default)
    b.sine1([1,0.5,0.33,0.25,0.2], true, true, true);
```

```
)

    b.sine1([1], true, true, true);
// can also be written as:
    b.sine1(1.0/[1,2,3,4,5], true, true, true);
// can also be written as:
    b.sine1(1.0/(1..12));
    b.sine1(1.0/(1..24));
    b.sine1(1.0/(1..32));

// waveform buffer has values and adjacent differences stored inside
    b.plot
```

Osc
arguments: buffer number, frequency, phase, mul, add.

```
{ Osc.ar(b.bufnum, 400, 0, 0.5) }.scope(1, zoom:0.5);
 // aliasing?
{ Osc.ar(b.bufnum, XLine.kr(200,20000,10), 0, 0.5) }.scope(1, zoom:0.5);
```

Try:

```
    b.sine1([1, 0, 0.5, 0, 0.33, 0, 0.25, 0, 0.2,],true, true, true);
```

COsc - two oscillators, detuned
arguments: buffer number, frequency, beat frequency, mul, add.

```
{ COsc.ar(b.bufnum, 400, 0.4, 0.3) }.scope(1, zoom:0.5)
// change buffer as above.

{ COsc.ar(b.bufnum, 400, MouseX.kr(0.0,80.0,'linear'), 0.3) }.scope(1, zoom:
0.5);
```

Also,

```
b.sine1([1], true, true, true);
```

VOsc - multiple wave table crossfade oscillators
arguments: buffer number, frequency, phase, mul, add.

```
(
    // allocate buffers 80 to 87
    b = Array.new(8);
```

```
    8.do {arg i; b = b.add(Buffer.alloc(s, 2048, 1, bufnum: 80+i))};
)

(
    // fill buffers 80 to 87
    8.do({arg i;
        var n, a;
        // generate array of harmonic amplitudes
        n = (i+1)**2; // num harmonics for each table: [1,4,9,16,25,36,49,64]
        a = {arg j; ((n-j)/n).squared }.dup(n);
        // fill table
        b[i].sine1(a);
    });
)

// Now, do it
{ VOsc.ar(MouseX.kr(80,87), 120, 0, 0.5) }.scope(1, zoom:2);
```

VOsc3 - three VOscs summed.
arguments: buffer number, freq1, freq2, freq3, beat frequency, mul, add.

```
// chorusing
{ VOsc3.ar(MouseX.kr(80,87), 120, 121.04, 119.37, 0.2) }.scope(1, zoom:2);

// chords
{ VOsc3.ar(MouseX.kr(80,87), 120, 151.13, 179.42, 0.2) }.scope(1, zoom:2);
```

LF - "Low Frequency" Unit Generators.

LFTri, Impulse, LFSaw, LFPulse, VarSaw, SyncSaw

geometric waveforms, not band limited.

LFTri, LFSaw, Impulse
arguments: frequency, phase, mul, add

```
{ LFTri.ar(200,0,0.5) }.scope(1, zoom:0.5);
{ LFSaw.ar(200,0,0.5) }.scope(1, zoom:0.5);
{ Impulse.ar(200,0,0.5) }.scope(1, zoom:0.5);
```

```
{ Impulse.ar(MouseX.kr(0.1,1000),0,0.5) }.scope(1, zoom:0.5);

 // aliasing?
{ LFTri.ar(XLine.kr(200,20000,6), 0, 0.5) }.scope(1, zoom:0.5);
```

LFPulse, VarSaw

arguments: frequency, phase, width, mul, add

```
{ LFPulse.ar(200,0,0.3,0.5) }.scope(1);
{ VarSaw.ar(200,0,0.3,0.5) }.scope(1);
// pulse width modulation
{ LFPulse.ar(200,0,MouseX.kr(0,1),0.5) }.scope(1);
{ VarSaw.ar(200,0,MouseX.kr(0,1),0.5) }.scope(1);
```

SyncSaw

arguments: syncFreq, sawFreq, mul, add

A sawtooth wave that is hard synched to a fundamental pitch. This produces an effect similar to moving formants or pulse width modulation. The sawtooth oscillator has its phase reset when the sync oscillator completes a cycle. This is not a band limited waveform, so it may alias.

```
{ SyncSaw.ar(100, MouseX.kr(100, 1000), 0.5) }.scope(1, zoom: 4);
```