Env : Object

Specification for a segmented envelope

Source: Env.sc
Subclasses: Penv

See also: EnvGen, IEnvGen, Pseg

Description

An Env is a specification for a segmented envelope. Envs can be used both server-side, by an EnvGen or an IEnvGen within a SynthDef, and clientside, with methods such as -at and -asStream, below.

An Env can have any number of segments which can stop at a particular value or loop several segments when sustaining. It can have several shapes for its segments.

The envelope is conceived as a sequence of *nodes* (not to be confused with a synthesis-Node) : the first node gives the initial level of the envelope, and the following have three parameters: a target level, a time duration from the previous node, and a shape. The three parameters for each node are kept in separate arrays as explained below.

```
Env.new(levels: [0, 1, 0.9, 0], times: [0.1, 0.5, 1], curve: [-5, 0, -5]).plot;
```

In this envelope, there are four *nodes* :

- the first *node* is the initial level of the envelope : 0
- the second *node* has level 1 and is reached in 0.1 second
- the third *nodes* has level 0.9 and is reached in 0.5 second
- the fourth *nodes* has level 0 and is reached in 1 second

Close attention must be paid when retriggering envelopes. Starting from their value at the moment of retrigger, envelopes will cycle through all of their nodes, with the exception of the first. The first node is an envelope's initial value and is only output prior to the initial trigger.

```
(
{
    EnvGen.kr(
        Env(
            levels: [0, 0.1, 0.2, 0.3],
            times: [0.1, 0.1, 0.1],
            curve: 8
        ),
        gate: Impulse.kr(3)
    );
}.plot(duration: 1);
)
```

In the above example, the initial level (0) is never repeated. When retriggered, the envelope moves from its current value (0.3), to the value of the second node (0.1), and so forth.

NOTE: In some situations we deal with control points or breakpoints. If these control points have associated x positions (say in an envelope GUI, see EnvelopeView) they must be converted to time differences between points to be used as nodes in a Env object. The methods *xyc and *pairs can be used to specify an envelope in terms of points.

```
// an envelope in a synth
(
{
    var env = Env([0, 1, 0.5, 1, 0], [0.01, 0.5, 0.02, 0.5]);
    SinOsc.ar(470) * EnvGen.kr(env, doneAction: Done.freeSelf)
}.play
)
// an envelope to control a parameter in a pattern
(
Pbind(
    \note,  Env([0, 12, 6, 13, 0], [1, 5, 2, 10]),
    \dur, 0.1
).play
)
```

Class Methods

```
Env.new(levels: [ 0, 1, 0 ], times: [ 1, 1 ], curve: 'lin', releaseNode,
      loopNode, offset: 0)
```

Create a new envelope specification.

Arguments:

levels an array of levels. The first value is the initial level of the envelope. When the envelope is used with an EnvGen, levels can be any UGen (new level values are updated only when the envelope has reached that point). When the array of levels contains itself an array, the envelope returns a multichannel output (for a discussion, see Multichannel expansion)

times an array of durations of segments in seconds. There should be one fewer duration than there are levels, but if shorter, the array is extended by wrapping around the given values.

curve a Symbol, Float, or an Array of those. Determines the shape of the envelope segments.

The possible values are:

\step		flat segments (immediately jumps to final value)
\hold		flat segments (holds initial value, jump to final value at the end of the segment)
\linear	\lin	linear segments, the default
\exponential	\exp	natural exponential growth and decay. In this case, the levels must all be nonzero and have the same sign.
\sine	\sin	sinusoidal S shaped segments.
\welch	\wel	sinusoidal segments shaped like the sides of a Welch window.
\squared	\sqr	squared segment
\cubed	\cub	cubed segment
		a curvature value for all segments. 0 means linear, positive and

a Float		negative numbers curve the segment up and down.
an Array of symbols or floats		curvature values for each segment.

releaseNode an Integer or nil. The envelope will sustain at the releaseNode until released.

```
(
{
    EnvGen.kr(
        Env.new(
            levels: [0, 1, 0.5, 0],
            times: [0.01, 0.01, 0.01],
            releaseNode: 2 // sustains at level 0.5 until gate
is closed
        ),
        gate: Trig.kr(Impulse.kr(3), dur: 0.3)
    );
}.plot(duration: 1);
)
```

In the above example, the release node is set to the third node, which means it will sustain at the level of 0.5 until it is released. The envelope will then continue on until its last node is reached.

loopNode an Integer or nil. Creates a segment of looping nodes. You must specify a releaseNode in order for loopNode to have any effect. The loopNode is the initial node of the loop and is never repeated. Upon reaching the releaseNode, the envelope will move back to the node that immediately follows loopNode. The envelope will loop until its gate is closed. When released, a looping envelope will move from its current position to the node that immediately follows releaseNode and continue until the end.

```
(
{
    EnvGen.kr(
        Env([0, 1, 0, 0.2, 0, 0.5, 0.8, 0], [0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01], releaseNode: 5, loopNode: 1),
        gate: Trig.kr(1, 0.9)
    );
}.plot(duration: 1)
)
```

In this example :

- the starting level of the envelope is 0
- the loop goes from nodes[5] (value : 0.5) to the nodes[1+1] (value : 0)
- at time = 0.9, the loop is released, so the envelope goes to nodes[5+1] (value : 0.8)

offset an offset to all time values (only applies in IEnvGen).

Discussion:

```
(
{
    var env = Env([0.0, 0.5, 0.0, 1.0, 0.9, 0.0], [0.05, 0.1, 0.01, 1.0, 1.5],
```

```
-4);
    var envgen = EnvGen.ar(env, doneAction: Done.freeSelf);
    SinOsc.ar(
        envgen * 1000 + 440
    ) * envgen * 0.1
}.play
);
```

Env.**newClear**(numSegments: 8, numChannels: 1)

Creates a new envelope specification with **numSegments** and **numChannels** for filling in later.

Discussion:

This can be useful when passing Env parameters as args to a Synth. Note that the maximum number of segments is fixed and cannot be changed once embedded in a SynthDef. Trying to set an Env with more segments than then this may result in other args being unexpectedly set.

```
(
SynthDef(\help_Env_newClear, { |out = 0, gate = 1|
    var env, envctl;
    // make an empty 4 segment envelope
    env = Env.newClear(4);
    // create a control argument array
    envctl = \env.kr(env.asArray);
    Out.ar(out, SinOsc.ar(EnvGen.kr(envctl, gate), 0) * -12.dbamp);
}).add;
)

Synth(\help_Env_newClear, [\env, Env([700,900,900,800], [1,1,1], \exp)]); // 3
segments

// reset then play again:
Synth(\help_Env_newClear, [ \env, Env({ rrand(60, 70).midicps } ! 4, [1,1,1],
\exp)]);

// the same written as an event:
(instrument: \help_Env_newClear, env: Env({ rrand(60, 70).midicps } ! 4,
[1,1,1], \exp)).play;
```

Env.**shapeNames**

returns the dictionary containing the available shapes for the envelopes' curves

Env.**shapeNumber**(shapeName)

returns the index in the dictionary of the given curve shape

Arguments:

shapeName name of the shape. e.g. \lin, \cub ...

Standard Shape Envelope Creation Methods

The following class methods create some frequently used envelope shapes based on supplied durations.

```
Env.linen(attackTime: 0.01, sustainTime: 1.0, releaseTime: 1.0, level: 1.0,
      curve: 'lin')
```

Creates a new envelope specification which has a trapezoidal shape.

Arguments:

attackTime	the duration of the attack portion.
sustainTime	the duration of the sustain portion.
releaseTime	the duration of the release portion.
level	the level of the sustain portion.
curve	the curvature of the envelope.

Discussion:

```
Env.linen(0.1, 0.2, 0.1, 0.6).test.plot;
Env.linen(1, 2, 3, 0.6).test.plot;
Env.linen(1, 2, 3, 0.6, \sine).test.plot;
Env.linen(1, 2, 3, 0.6, \welch).test.plot;
Env.linen(1, 2, 3, 0.6, -3).test.plot;
Env.linen(1, 2, 3, 0.6, -3).test.plot;
Env.linen(1, 2, 3, 0.6, [[\sine, \welch, \lin, \exp]]).plot;
```

```
Env.triangle(dur: 1.0, level: 1.0)
```

Creates a new envelope specification which has a triangle shape.

Arguments:

dur	the duration of the envelope.
level	the peak level of the envelope.

Discussion:

```
Env.triangle(1, 1).test.plot;
```

```
Env.sine(dur: 1.0, level: 1.0)
```

Creates a new envelope specification which has a hanning window shape.

Arguments:

dur	the duration of the envelope.
level	the peak level of the envelope.

Discussion:

```
Env.sine(1, 1).test.plot;
```

Env.**perc**(attackTime: 0.01, releaseTime: 1.0, level: 1.0, curve: -4.0)

Creates a new envelope specification which (usually) has a percussive shape.

Arguments:

attackTime	the duration of the attack portion.
releaseTime	the duration of the release portion.
level	the peak level of the envelope.
curve	the curvature of the envelope.

Discussion:

```
Env.perc(0.05, 1, 1, -4).test.plot;
Env.perc(0.001, 1, 1, -4).test.plot;    // sharper attack
Env.perc(0.001, 1, 1, -8).test.plot;    // change curvature
Env.perc(1, 0.01, 1, 4).test.plot;    // reverse envelope
```

Env.**pairs**(pairs, curve)

Creates a new envelope specification from coordinates / control points

Arguments:

pairs	an array of pairs [[time, level], ...] if possible, pairs are sorted regarding their point in time
curve	the curvature of the envelope.

Discussion:

```
Env.pairs([[0, 1], [2.1, 0.5],  [3, 1.4]], \exp).plot;
Env.pairs([[0, 1], [3, 1.4], [2.1, 0.5], [3, 4]], \exp).plot; // *if
possible*, pairs are sorted according to time
Env.pairs({ { 1.0.rand } ! 2 } ! 16, \exp).plot;
```

Env.**xyc**(xyc)

Creates a new envelope specification from coordinates / control points with curvature.

Arguments:

xyc	an array of triplets [[time, level, curve], ...] if possible, pairs are sorted regarding their point in time

Discussion:

```
Env.xyc([[0, 1, \sin], [2.1, 0.5, \lin],  [3, 1.4, \lin]]).plot;
```

```
Env.xyc([[2.1, 0.5, \lin], [0, 1, \sin], [3, 1.4, \lin]]).plot; // *if
possible*, pairs are sorted according to time
Env.xyc({ [1.0.rand, 10.0.rand, -4.rand2] } ! 16, \exp).plot;
Env.xyc([[0, 1], [2.1, 0.5],  [3, 1.4]]).plot; // if not specified, curve
defaults to \lin
```

Sustained Envelope Creation Methods

The following methods create some frequently used envelope shapes which have a sustain segment. They are typically used in SynthDefs in situations where at the time of starting the synth it is not known when it will end. Typical cases are external interfaces, midi input, or quickly varying TempoClock.

```
(
SynthDef(\env_help, { |out, gate = 1, amp = 0.1, release = 0.1|
    var env = Env.adsr(0.02, release, amp);
    var gen = EnvGen.kr(env, gate, doneAction: Done.freeSelf);
    Out.ar(out, PinkNoise.ar(1 ! 2) * gen)
}).add
);

a = Synth(\env_help);
b = Synth(\env_help, [\release, 2]);
a.set(\gate, 0); // alternatively, you can write a.release;
b.set(\gate, 0);
```

`Env.step(levels: [ 0, 1 ], times: [ 1, 1 ], releaseNode, loopNode, offset: 0)`

Creates a new envelope specification where all the segments are horizontal lines. Given n values of times only n levels need to be provided, corresponding to the fixed value of each segment.

Arguments:

levels	an array of levels. Levels can be any UGen (new level values are updated only when the envelope has reached that point). When the array of levels contains itself an array, the envelope returns a multichannel output (for a discussion, see Multichannel expansion)
times	an array of durations of segments in seconds. It should be the same size as the levels array.
releaseNode	an Integer or nil. The envelope will sustain at the release node until released.
loopNode	an Integer or nil. If not nil the output will loop through those nodes starting at the loop node to the node immediately preceding the release node, before back to the loop node, and so on. Note that the envelope only transitions to the release node when released. Examples are below. The loop is escaped when a gate signal is sent, when the output transitions to the release node, as described below.
offset	an offset to all time values (only applies in IEnvGen).

Discussion:

```
(
{
    var env = Env.step([0, 3, 5, 2, 7, 3, 0, 3, 4, 0], [0.5, 0.1, 0.2, 1.0,
1.5, 2, 0.2, 0.1, 0.2, 0.1]);
    var envgen = EnvGen.kr(env);
```

```
    var freq = (envgen + 60).midicps;
    SinOsc.ar(freq) * 0.1
}.play
);
```

Env.**adsr**(attackTime: 0.01, decayTime: 0.3, sustainLevel: 0.5, releaseTime: 1.0, peakLevel: 1.0, curve: -4.0, bias: 0.0)

Creates a new envelope specification which is shaped like traditional analog attack-decay-sustain-release (adsr) envelopes.

Arguments:

attackTime	the duration of the attack portion.
decayTime	the duration of the decay portion.
sustainLevel	the level of the sustain portion as a ratio of the peak level.
releaseTime	the duration of the release portion.
peakLevel	the peak level of the envelope.
curve	the curvature of the envelope.
bias	offset

Discussion:

```
Env.adsr(0.02, 0.2, 0.25, 1, 1, -4).test(2).plot;
Env.adsr(0.001, 0.2, 0.25, 1, 1, -4).test(2).plot;
Env.adsr(0.001, 0.2, 0.25, 1, 1, -4).test(0.45).plot;    // release after 0.45
sec
```

Env.**dadsr**(delayTime: 0.1, attackTime: 0.01, decayTime: 0.3, sustainLevel: 0.5, releaseTime: 1.0, peakLevel: 1.0, curve: -4.0, bias: 0.0)

As *adsr above, but with its onset delayed by **delayTime** in seconds. The default delay is 0.1.

Env.**asr**(attackTime: 0.01, sustainLevel: 1.0, releaseTime: 1.0, curve: -4.0)

Creates a new envelope specification which is shaped like traditional analog attack-sustain-release (asr) envelopes.

Arguments:

attackTime	the duration of the attack portion.
sustainLevel	the level of the sustain portion as a ratio of the peak level.
releaseTime	the duration of the release portion.
curve	the curvature of the envelope.

Discussion:

```
Env.asr(0.02, 0.5, 1, -4).test(2).plot;
Env.asr(0.001, 0.5, 1, -4).test(2).plot; // sharper attack
Env.asr(0.02, 0.5, 1, 'linear').test(2).plot; // linear segments
```

Env.**cutoff**(releaseTime: 0.1, level: 1.0, curve: 'lin')

Creates a new envelope specification which has no attack segment. It simply sustains at the peak level until released. Useful if you only need a fadeout, and more versatile than Line.

Arguments:

releaseTime	the duration of the release portion.
level	the peak level of the envelope.
curve	the curvature of the envelope.

Discussion:

```
Env.cutoff(1, 1).test(2).plot;
Env.cutoff(1, 1, 4).test(2).plot;
Env.cutoff(1, 1, \sine).test(2).plot;
```

Env.**circle**(levels, times, curve: 'lin')

Creates a new envelope specification which cycles through its values. For making a given envelope cyclic, you can use the instance method -circle

Arguments:

levels	The levels through which the envelope passes.
times	The time between subsequent points in the envelope, which may be a single value (number), or an array of them. If too short, the array is extended. In difference to the *new method, the size of the times array is the same as that of the levels, because it includes the loop time.
curve	The curvature of the envelope, which may be a single value (number or symbol), or an array of them. If too short, the array is extended. In difference to the *new method, the size of the curve array is the same as that of the levels, because it includes the loop time.

Discussion:

```
{ SinOsc.ar(EnvGen.kr(Env.circle([0, 1, 0], [0.01, 0.5, 0.2])) * 440 + 200) *
0.2 }.play;
{ SinOsc.ar(EnvGen.kr(Env.circle([0, 1, 0, 2, 0, 1, 0], [0.01, 0.3])) * 440 +
200) * 0.2 }.play;
{ SinOsc.ar(EnvGen.kr(Env.circle([0, 1, 0, (2..4), 0, (1..3), 0], [0.01,
0.3])) * 440 + 200).sum * 0.2 }.play; // multichannel expanded levels
```

Multichannel expansion

If one of the values within either levels, times, or curves is itself an array, the envelope expands to multiple channels wherever appropriate. This means that when such an envelope is passed to an EnvGen, this EnvGen will expand,

and when the envelope is queried via the methods -at or -asSignal, it will return an array of values.

```
(
{
    var env = Env([0.0, 0.5, 0.0, [1.0, 1.25, 1.5], 0.9, 0.0], [0.05, 0.1, 0.01,
1.0, 1.5], -4);
    var envgen = EnvGen.ar(env, doneAction: Done.freeSelf);
    SinOsc.ar(
        envgen * 1000 + 440
    ) * envgen * 0.1
}.play
);

(
{
    var env = Env([1, [1, 2, 3], 0.5, 0.5, [3, 2, 1], 2], [1, 1, 0.5, 1], [[\exp,
\sin]]);
    env.plot;
    Splay.ar(SinOsc.ar(EnvGen.kr(env) * 400 + 600)) * 0.1
}.play;
);


(
{
    var levels = (1..30);
    var env = Env([1, levels, 0.5, levels / 2.5, 2], [1, 0.15, 1, 0.25, 0.1],
\exp);
    Splay.ar(SinOsc.ar(EnvGen.kr(env) * 400 + 600)) * 0.1
}.play;
);


// accessing the envelope by indexing

e = Env([1, [1, 2, 3], 1], [1, 1], \exp);
e.at(0.5);
e.at(1.8);
e.at(2);

e = Env([1, 1, 1], [1, [1, 2, 3]], \exp);
e.at(0.5);
e.at(2);


// multichannel levels

Env([0.1, 1, 0.1], [1, [1, 2, 3]], \exp).plot;
Env([0.1, 1, 0.1], [1, [1, 2, 3]], [\lin, [\lin, \exp, \sin]]).plot;

Env([1, 1, 0.5, 3, 2], [1, 0.5, 1, 0.25], \exp).plot;
Env([0, 1, 0, 2, 0] * [[1, 2, 3]], [1, 0.5, 1, 0.25], \lin).plot;


// multichannel curves
```

```
Env([0.01, 5, 1, 0.5] + 1, [1, 0.5, 1, 0.25], [[\lin, \sqr]]).plot;

Env([0.01, 5, 1, 0.5, 0.001] + 1, [1, 0.5, 1, 0.25, 1], [[\lin, \cub, \sin,
\cubed, \welch, \step, \exp]]).plot(bounds: Rect(30, 100, 500, 700));

Env([0.01, 5, 1, 0.5, 0.001] + 1, [1, 0.5, 1, 0.25, 1], [(-4..4)]).plot(bounds:
Rect(30, 100, 500, 700));
Env([0.01, 5, 1, 0.5] + 1, [1, 0.5, 1, 0.25], [(-4..4)]).plot(bounds: Rect(30,
100, 500, 700));


Env([[0, 0.01], 1, 0], [0.5, 0.5], [[\lin, \exp], \step]).plot;
Env([[0, 0.01], 1, [0, 0.01]], [0.5, 1], [[\lin, \exp]]).plot;

// multichannel times

Env([[2, 1], 0], [[1, 2]], \lin).plot;
Env([0, 1], [1/(1..5)], [(-4..4)]).plot(bounds: Rect(30, 100, 300, 700));
Env([0, 1], [1/(1..5)], \lin).plot(bounds: Rect(30, 100, 300, 700));


// mixed expansions

Env([1, [ 1, 2, 3, 4, 5 ], 0.5, [3, 2, 1], 2], [1, 0.5, 1, 0.25], [[\exp,
\lin]]).plot;
Env([1, [ 1, 2, 3, 4, 5 ], 0.5, 4, 2], [1, 0.5, 1, 0.25], \exp).plot;


// expanding control point envelopes

Env.xyc([[2, 0.5, [\lin, \exp]], [0, 1, \lin], [3, 1.4, \lin]]).plot;
Env.xyc({ [1.0.rand, 1.0.rand, {[\lin, \exp, \step].choose} ! 3] } ! 8).plot

Env.xyc([[[2.0, 2.3], 0.5, \lin], [0, 1, \lin], [3, 1.4, \lin]]).plot; //
multiple times
```

Inherited class methods

7 methods from Object ► show

Instance Methods

.ar(doneAction: 0, gate: 1.0, timeScale: 1.0, levelScale: 1.0, levelBias: 0.0)

.kr(doneAction: 0, gate: 1.0, timeScale: 1.0, levelScale: 1.0, levelBias: 0.0)

Instead of using an EnvGen inside a UGen graph, this message does the same implicitly for convenience. Its argument order corresponds to the most common arguments.

Arguments:

doneAction An integer representing an action to be executed when the env is finished playing. This can be used to free the enclosing synth, etc. See Done for more detail.

gate This triggers the envelope and holds it open while > 0. If the Env is fixed-length (e.g. Env.linen, Env.perc), the gate argument is used as a simple trigger. If it is an sustaining envelope (e.g. Env.adsr, Env.asr), the envelope is held open until the gate becomes 0, at which point is released.

If **gate** < 0, force release with time `-1.0 - gate`. See EnvGen: Forced release example.

timeScale The durations of the segments are multiplied by this value. This value can be modulated, but is only sampled at the start of a new envelope segment.

levelScale The levels of the breakpoints are multiplied by this value. This value can be modulated, but is only sampled at the start of a new envelope segment.

levelBias This value is added as an offset to the levels of the breakpoints. This value can be modulated, but is only sampled at the start of a new envelope segment.

Discussion:

```
{ Blip.ar(50, 200, Env.perc(1, 0.1, 0.2).kr(2)) }.play;
(
{
    Blip.ar(
        Env({ exprand(3, 2000.0) } ! 18, 0.2, \exp).kr,
        200,
        Env({ rrand(0.1, 0.2) } ! 18 ++ 0, 0.2).kr(2))
    }.play;
)
```

.blend(argAnotherEnv, argBlendFrac: 0.5)

Blend two envelopes. Returns a new Env. See blend example below.

Arguments:

argAnotherEnv an Env.

argBlendFrac a number from zero to one.

.delay(delay)

Returns a new Env based on the receiver in which the start value will be held for **delay** number of seconds.

Arguments:

delay The amount of time to delay the start of the envelope.

Discussion:

```
a = Env.perc(0.05, 1, 1, -4);
b = a.delay(2);
a.test.plot;
b.test.plot;

a = Env([0.5, 1, 0], [1, 1]).plot;
a.delay(1).plot;
```

.duration

.duration = dur

Set the total duration of times, by stretching them.

Discussion:

```
e = Env([0, 1, 0], [1, 2]);
e.duration;
e.duration = 2;
e.duration;
```

.totalDuration

Get the total duration of the envelope. In multi-channel envelopes, this is the duration of the longest one.

Discussion:

```
e = Env([0, 1, 0], [[1, 2], 2]);
e.duration;
e.totalDuration;
```

.circle(timeFromLastToFirst: 0.0, curve: 'lin')

circle from end to beginning over the time specified, with the curve specified. See also the class method *circle

Discussion:

```
(
{ SinOsc.ar(
    EnvGen.kr(
        Env([6000, 700, 100], [1, 1], ['exp', 'lin']).circle.postcs)
    ) * 0.1
    + Impulse.ar(1) }.play;
)

(
{ SinOsc.ar(
    EnvGen.kr(
        Env([6000, 700, 100], [1, 1], ['exp', 'lin']).circle(1).postcs,
        MouseX.kr > 0.5)
    ) * 0.1
    + Impulse.ar(1) }.play;
)
```

.test(releaseTime: 3.0)

Test the envelope on the default Server with a SinOsc.

Arguments:

releaseTime If this is a sustaining envelope, it will be released after this much time in seconds. The default is 3 seconds.

.plot(size: 400, bounds, minval, maxval, name)

From extension in /usr/local/share/SuperCollider/SCClassLibrary/Common/GUI/PlusGUI/Math/PlotView.sc

Plot this envelope's shape in a window.

Arguments:

size The size of the plot. The default is 400.

bounds the size of the plot window.

minval the minimum value in the plot. Defaults to the lowest value in the data.

maxval the maximum value in the plot. Defaults to the highest value in the data.

name the plot window's label name. If nil, a name will be created for you.

.asSignal(length: 400)

Returns a Signal of size **length** created by sampling this Env at **length** number of intervals. If the envelope has multiple channels (see Multichannel expansion), this method returns an array of signals.

.asArray

Converts the Env to an Array in a specially ordered format. This allows for Env parameters to be settable arguments in a SynthDef. See example under *newClear.

.asMultichannelArray

Converts the Env to an Array in a specially ordered format, like -asArray, however it always returns an array of these data sets, corresponding to the number of channels of the envelope.

.isSustained

Returns true if this is a sustaining envelope, false otherwise.

.range(lo: 0.0, hi: 1.0)

.exprange(lo: 0.01, hi: 1.0)

.curverange(lo: 0.0, hi: 1.0, curve: -4)

Returns a copy of the Env whose levels have been mapped onto the given linear, exponential or curve range.

Discussion:

```
a = Env.adsr;
a.levels;
a.range(42, 45).levels;
a.exprange(42, 45).levels;
a.curverange(42, 45, -4).levels;

(
// Mapping an Env to an exponential frequency range:
{
    SinOsc.ar(EnvGen.ar(Env.perc(0.01, 0.7).exprange(40, 10000), doneAction:
Done.freeSelf)) * 0.2;
}.play
)
```

Client-side Access and Stream Support

Sustain and loop settings have no effect in the methods below.

.at(time)

Returns the value of the Env at **time**. If the envelope has multiple channels, this method returns an array of levels.

Arguments:

time A number or an array of numbers to specify a cut in the envelope. If time is an array, it returns the corresponding levels of each time value, and if the envelope has multiple channels, it returns an array of values. A combination of both returns a two-dimensional array.

Discussion:

```
e = Env.triangle(1, 1);
e.at(0.5);
e.at([0.5, 0.7]);

e = Env([1, [1, 2, 3], 1], [1, 1], \exp);
e.at(0.5);
e.at(1.8);
e.at(2);
e.at([0.5, 1.2]);

e = Env([1, 100, 1], [1, [1, 2, 3]], \exp);
e.at(0.5);
e.at(2);
e.at([1, 2, 4]);
```

.embedInStream(inval)

Embeds this Env within an enclosing Stream. Timing is derived from `thisThread.beats`.

.asStream

Creates a Routine and embeds the Env in it. This allows the Env to function as a Stream.

Discussion:

```
(
{
e = Env.sine.asStream;
5.do({
    e.next.postln;
    0.25.wait;
})}.fork
)
```

Inherited instance methods

262 methods from Object ▶ show

Undocumented instance methods

==(that)

.array

.asArrayForInterpolation

.asControlInput

From extension in /usr/local/share/SuperCollider/SCClassLibrary/Common/Control/extConvertToOSC.sc

.asMultichannelSignal(length: 400, class)

.asOSCArgEmbeddedArray(array)

From extension in /usr/local/share/SuperCollider/SCClassLibrary/Common/Control/extConvertToOSC.sc

.asPseg

.curveValue(curve)

.curves

.curves = z

.discretize(n: 1024)

.hash

.levels

.levels = z

.loopNode

.loopNode = z

.offset

.offset = z

.releaseNode

.releaseNode = z

.releaseTime

.times

.times = z

Examples

```
s.boot;     //.test below will run a synthesis example
        // to demonstrate the envelope, so the Server must be on

// different shaped segments: .plot graphs the Env
Env.new([0,1, 0.3, 0.8, 0], [2, 3, 1, 4],'linear').test.plot;
Env.new([0.001, 1, 0.3, 0.8, 0.001], [2, 3, 1, 4],'exponential').test.plot;
Env.new([0, 1, 0.3, 0.8, 0], [2, 3, 1, 4],\sine).test.plot;
Env.new([0.001, 1, 0.3, 0.8, 0.001],[2,3,1,4],\welch).test.plot;
Env.new([0, 1, 0.3, 0.8, 0], [2, 3, 1, 4],'step').test.plot;
Env.new([0, 1, 0.3, 0.8, 0], [2, 3, 1, 4], -2).test.plot;
Env.new([0, 1, 0.3, 0.8, 0], [2, 3, 1, 4], 2).test.plot;
Env.new([0, 1, 0.3, 0.8, 0], [2, 3, 1, 4], [0, 3, -3, -1]).test.plot;
```

If a release node is given, and the gate input of the EnvGen is set to zero, it outputs the nodes after the release node:

```
// release node is node 1; takes 0.5 seconds to go from 0 to 1,
// sustains at level of 1, then released after three seconds
// (test causes the release after three seconds, given the argument 3),
// taking 2 seconds to finish
Env.new([0,1,0],[0.5,2],'linear',1).test(3).plot

// more complex examples
// release node is node 2; releases after 5 sec
Env.new([0.001,1,0.3,0.8,0.001],[2,3,1,4] * 0.2, 2, 2).test(5).plot;
Env.new([0.001,1,0.3,0.8,0.5,0.8,0],[2,3,1,2,2,1] * 0.2, 2, 2).test(5).plot;
```

```
// early release: goes straight onto the release node after 0.1 seconds
Env.new([0.001,1,0.3,0.8,0.5,0.8,0],[2,3,1,2,2,1] * 0.2, 2, 2).test(0.1).plot;
```

If a loop node is given, the EnvGen outputs the nodes between the loop node and the release node (not including the release node itself) until it is released:

```
// release node is node 2, loop node is node 0: so loops around nodes 0 (lvl 1,
dur 0.5)
// and 1 (lvl 0.1, dur 0.5)         //until released after 3.5 seconds
Env.new([0,1,0.1,0],[0.5,0.5,2], 'lin', 2, 0).test(3.5).plot;

// this just sustains at node 0, because there is no other node to loop around!
Env.new([0,1,0],[0.5,2], 'lin', 1, 0).test(3.5).plot;

// more complex example: release node is node 3, loop node is node 1
Env.new([0.001,1,0.3,0.8,0.5,0.8,0],[2,1,1,2,3,1] * 0.1, 'lin', 3,
1).test(3).plot;

// this is the resulting graph:
(
e = Env.new([0.001,1,0.3,0.8,0.5,0.8,0],[2,1,1,2,3,1] * 0.001, 'lin', 3, 1);
e.plot;{ EnvGen.ar(e, Trig.ar(Impulse.ar(0), 10*0.001)) }.plot(0.02);
)
```

NOTE: The starting level for an envelope segment is always the level you are at right now. For example when the gate is released and you jump to the release segment, the level does not jump to the level at the beginning of the release segment, it changes from whatever the current level is to the goal level of the release segment over the specified duration of the release segment.

There is an extra level at the beginning of the envelope to set the initial level. After that each node is a goal level and a duration, so node zero has duration equal to times[0] and goal level equal to levels[1].

The loop jumps back to the loop node. The endpoint of that segment is the goal level for that segment and the duration of that segment will be the time over which the level changed from the current level to the goal level.

blend

```
a = Env([0, 0.2, 1, 0.2, 0.2, 0], [0.5, 0.01, 0.01, 0.3, 0.2]);
a.test.plot;

b = Env([0, 0.4, 1, 0.2, 0.5, 0], [0.05, 0.4, [0.01, 0.1], 0.1, 0.4]);
b.test.plot;

(
Task({
    f = (0, 0.2 .. 1);
    f.do { |u|
        blend(a, b, u).test.plot;
        2.wait;
        Window.allWindows.pop.close; // close last opened window
    }
}).play(AppClock);
```

```
)

// blend in a SynthDef
(
SynthDef(\help_EnvBlend, { | out, factor = 0 |
    Out.ar(out, EnvGen.kr(blend(Env.perc, Env.sine, factor), 1.0, doneAction:
Done.freeSelf)
        * SinOsc.ar(440,0,0.1)
    )
}).add
);

(
{
    var factors = (0, 0.1..1);
    factors.do {|f| Synth(\help_EnvBlend, [\factor, f.postln]); 1.wait };
}.fork
);
```

helpfile source: /usr/local/share/SuperCollider/HelpSource/Classes/Env.schelp
link::Classes/Env::