DaisySP

Contents

adenv

Author: shensley

Trigger-able envelope with adjustable min/max, and independent per-segment time control.

TODO:

- Add Cycling
- Implement Curve (its only linear for now).
- Maybe make this an AD_sr that has AD/AR/A_sr modes.

Envelope Segments

Distinct stages that the phase of the envelope can be located in.

- IDLE = located at phase location 0, and not currently running
- ATTACK = First segment of envelope where phase moves from MIN value to MAX value
- DECAY = Second segment of envelope where phase moves from MAX to MIN value
- LAST = The final segment of the envelope (currently decay)

```
enum {
        ADENV_SEG_IDLE,
        ADENV_SEG_ATTACK,
        ADENV_SEG_DECAY,
        ADENV_SEG_LAST,
};
```

init

Initializes the ad envelope

float sample_rate - sample rate of the audio engine being run.

Defaults

- current segment = idle
- curve = linear
- phase = 0
- min = 0
- max = 1

```
void init(float sample_rate);
```

process

processes the current sample of the envelope. Returns the current envelope value. This should be called once per sample period.

```
float process();
```

trigger

Starts or retriggers the envelope.

```
inline void trigger() {_trigger = 1; }
```

Setters

set_segment_time

Sets the length of time(secondsVERIFYTHIS) for a specific segment.

```
inline void set_time(uint8_t seg, float time)
```

set_curve_scalar

Sets the amount of curve applied. Input range: -1 to 1. - At -1, curve = full logarithmic - At 1, curve = full exponential - At 0, curve = linear

```
inline void set_curve_scalar(float scalar) { _curve_scalar = scalar; }
```

set_min

Sets the minimum value of the envelope output Input range: -FLT_MAX, to FLT_MAX

```
inline void set_min(float min) {_min = min; }
```

set_max

Sets the maximum value of the envelope output Input range: -FLT_MAX, to FLT_MAX

```
inline void set_max(float max) {_max = max; }
```

Getters

current_segment

Returns the segment of the envelope that the phase is currently located in.

```
inline void current_segment();
```

compressor

influenced by compressor in soundpipe (from faust).

Modifications made to do:

- Less calculations during each process loop (coefficients recalculated on parameter change).
- C++-ified
- added sidechain support

TODO:

- With fixed controls this is relatively quick, but changing controls now costs a lot more
- Still pretty expensive
- Add soft/hard knee settings
- Maybe make stereo possible? (needing two for stereo is a bit silly, and their gain shouldn't be totally unique.

by: shensley

init

Initializes compressor

samplerate - rate at which samples will be produced by the audio engine.

```
void init(float samplerate);
```

process

compresses the audio input signal, either keyed by itself, or a secondary input.

in - audio input signal (to be compressed)

(optional) key - audio input that will be used to side-chain the compressor.

```
float process(float &in, float &key);
float process(float &in);
```

setters

set_ratio

amount of gain reduction applied to compressed signals

Expects 1.0 -> 40. (untested with values < 1.0)

```
void set_ratio(const float &ratio)
```

set_threshold

threshold in dB at which compression will be applied

Expects 0.0 -> -80.

```
void set_threshold(const float &thresh)
```

set_attack

envelope time for onset of compression for signals above the threshold.

Expects 0.001 -> 10

```
void set_attack(const float &atk)
```

set_release

envelope time for release of compression as input signal falls below threshold.

Expects 0.001 -> 10

```
void set_release(const float &rel)
```

crossfade

Performs a crossfade between two signals

Original author: Paul Batchelor

Ported from Soundpipe by Andrew Ikenberry added curve option for constant power, etc.

TODO:

- implement exponential curve process
- implement logarithmic curve process

Curve Options

Curve applied to the crossfade

- LIN = linear
- CPOW = constant power
- LOG = logarithmic
- EXP exponential
- LAST = end of enum (used for array indexing)

```
enum
{
    CROSSFADE_LIN,
    CROSSFADE_CPOW,
    CROSSFADE_LOG,
    CROSSFADE_EXP,
    CROSSFADE_LAST,
};
```

init

Initializes crossfade module

Defaults

- current position = .5
- curve = linear

```
inline void init()
```

process

processes crossfade and returns single sample

```
float process(float &in1, float &in2);
```

Setters

set_pos

Sets position of crossfade between two input signals

Input range: 0 to 1

```
inline void set_pos(float pos) { pos_ = pos; }
```

set_curve

Sets current curve applied to crossfade

Expected input: See Curve Options

```
inline void set_curve(uint8_t curve) { curve_ = curve; }
```

Getters

get_pos

Returns current position

```
inline float get_pos(float pos) { return pos_; }
```

get_curve

Returns current curve

```
inline uint8_t get_curve(uint8_t curve) { return curve_; }
```

dcblock

Removes DC component of a signal

init

Initializes dcblock module

```
void init(float sample_rate);
```

process

performs dcblock process

```
float process(float in);
```

decimator

Performs downsampling and bitcrush effects

init

Initializes downsample module

```
void init();
```

process

Applies downsample and bitcrush effects to input signal. Returns one sample. This should be called once per sample period.

```
float process(float input);
```

Setters

set_downsample_factor

Sets amount of downsample Input range:

```
inline void set_downsample_factor (float downsample_factor)
```

set_bitcrush_factor

Sets amount of bitcrushing Input range:

```
inline void set_bitcrush_factor (float bitcrush_factor)
```

set_bits_to_crush

Sets the exact number of bits to crush

0-16 bits

```
inline void set_bits_to_crush(const uint8_t &bits)
```

Getters

get_downsample_factor

Returns current setting of downsample

```
inline float get_downsample_factor () { return _downsample_factor; }
```

get_bitcrush_factor

Returns current setting of bitcrush

```
inline float get_bitcrush_factor () { return _bitcrush_factor; }
```

delayline

Simple Delay line.

November 2019

Converted to Template December 2019

declaration example: (1 second of floats)

```
delayline<float, SAMPLE_RATE> del;
```

By: shensley

init

initializes the delay line by clearing the values within, and setting delay to 1 sample.

```
void init()
```

reset

clears buffer, sets write ptr to 0, and delay to 1 sample.

```
void reset() {
```

set_delay

sets the delay time in samples

If a float is passed in, a fractional component will be calculated for interpolating the delay line.

```
inline void set_delay(size_t delay)
```

```
inline void set_delay(float delay)
```

write

writes the sample of type T to the delay line, and advances the write ptr

```
inline void write(const T sample)
```

read

returns the next sample of type T in the delay line, interpolated if necessary.

```
inline const T read() const
```

dsy_pstream.h Ported from Csound - October 2019

(c) Richard Dobson August 2001 NB pvoc routines based on CARL distribution(Mark Dolson). This file is licensed according to the terms of the GNU LGPL.

Definitions from Csound: pvsanal pvsfread pvsynth pvsadsyn pvscross pvsmaska = (overloaded, – should probably just pvsset(&src, &dest))

More or less for starting purposes we really only need pvsanal, and pvsynth to get started

line

creates a line segment signal

init

Initializes line module.

```
void init(float sample_rate);
```

process

Processes line segment. Returns one sample.

value of finished will be updated to a 1, upon completion of the line's trajectory.

```
float process(uint8_t *finished);
```

start

Begin creation of line.

Arguments:

- start - beginning value
- end - ending value
- dur - duration in seconds of line segment

```
void start(float start, float end, float dur);
```

metro

Creates a clock signal at a specific frequency.

init

Initializes metro module.

Arguments: - freq: frequency at which new clock signals will be generated Input Range: - sample_rate: sample rate of audio engine Input range:

```
void init(float freq, float sample_rate);
```

process

checks current state of metro object and updates state if necesary.

```
uint8_t process();
```

Setters

set_freq

Sets frequency at which metro module will run at.

```
void set_freq(float freq);
```

Getters

get_freq

Returns current value for frequency.

```
inline float get_freq() { return freq_; }
```

nlfilt

port by: stephen hensley, December 2019

Non-linear filter.

The four 5-coefficients: a, b, d, C, and L are used to configure different filter types.

Structure for Dobson/Fitch nonlinear filter

Revised Formula from Risto Holopainen 12 Mar 2004

`Y{n} =tanh(a Y{n-1} + b Y{n-2} + d Y^2{n-L} + X{n} - C)`

Though traditional filter types can be made, the effect will always respond differently to different input.

This Source is a heavily modified version of the original source from Csound.

TODO:

- make this work on a single sample instead of just on blocks at a time.

init

Initializes the nlfilt object.

```
void init();
```

process

Process the array pointed to by *in and updates the output to *out;

This works on a block of audio at once, the size of which is set with the size.

```
void process_block(float *in, float *out, size_t size);
```

setters

set_coefficients

inputs these are the five coefficients for the filter.

```
inline void set_coefficients(float a, float b, float d, float C, float L)
```

individual setters for each coefficients.

```
inline void set_a(float a) { a_ = a; }
```

```
inline void set_b(float b) { b_ = b; }
```

```
inline void set_d(float d) { d_ = d; }
```

```
inline void set_C(float C) { C_ = C; }
```

```
inline void set_L(float L) { L_ = L; }
```

oscillator

```
Synthesis of several waveforms, including polyBLEP bandlimited waveforms.
```

example:

```
daisysp::oscillator osc;
init()
{
    osc.init(SAMPLE_RATE);
    osc.set_frequency(440);
    osc.set_amp(0.25);
    osc.set_waveform(WAVE_TRI);
}

callback(float *in, float *out, size_t size)
{
    for (size_t i = 0; i < size; i+=2)
    {
        out[i] = out[i+1] = osc.process();
    }
}
```

Waveforms

Choices for output waveforms, POLYBLEP are appropriately labeled. Others are naive forms.

```
enum
{
    WAVE_SIN,
    WAVE_TRI,
    WAVE_SAW,
    WAVE_RAMP,
    WAVE_SQUARE,
    WAVE_POLYBLEP_TRI,
    WAVE_POLYBLEP_SAW,
    WAVE_POLYBLEP_SQUARE,
    WAVE_LAST,
};
```

init

Initializes the oscillator

float samplerate - sample rate of the audio engine being run, and the frequency that the process function will be called.

Defaults: - freq = 100 Hz - amp = 0.5 - waveform = sine wave.

```
void init(float samplerate)
```

set_freq

Changes the frequency of the oscillator, and recalculates phase increment.

```
inline void set_freq(const float f)
```

set_amp

Sets the amplitude of the waveform.

```
inline void set_amp(const float a) { amp = a; }
```

set_waveform

Sets the waveform to be synthesized by the process() function.

```
inline void set_waveform(const uint8_t wf) { waveform = wf < WAVE_LAST ? wf : WAVE_S
```

process

Processes the waveform to be generated, returning one sample. This should be called once per sample period.

```
float process();
```

phasor

Generates a normalized signal moving from 0-1 at the specified frequency.

TODO:

I'd like to make the following things easily configurable:

- Selecting which channels should be initialized/included in the sequence conversion.
- Setup a similar start function for an external mux, but that seems outside the scope of this file.

init

Initializes the phasor module

sample rate, and freq are in Hz

initial phase is in radians

Additional init functions have defaults when arg is not specified:

- phs = 0.0f
- freq = 1.0f

```
inline void init(float sample_rate, float freq, float initial_phase)
```

```
inline void init(float sample_rate, float freq)
```

```
inline void init(float sample_rate)
```

process

processes phasor and returns current value

```
float process();
```

Setters

set_freq

Sets frequency of the phasor in Hz

```
void set_freq(float freq);
```

Getters

get_freq

Returns current frequency value in Hz

```
inline float get_freq() { return freq_; }
```

pluck

Produces a naturally decaying plucked string or drum sound based on the Karplus-Strong algorithms.

This code has been extracted from the Csound opcode "pluck" It has been modified to work as a Daisy Soundpipe module.

Original Author(s): Barry Vercoe, John ffitch

Year: 1991

Location: OOps/ugens4.c

Mode

The method of natural decay that the algorithm will use.

- RECURSIVE: 1st order recursive filter, with coefs .5.
- WEIGHTED_AVERAGE: weighted averaging.

```
enum
{
    PLUCK_MODE_RECURSIVE,
    PLUCK_MODE_WEIGHTED_AVERAGE,
};
```

init

Initializes the Pluck module.

Arguments:

- sample_rate: Sample rate of the audio engine being run.
- buf: buffer used as an impulse when triggering the pluck algorithm
- npt: number of elementes in buf.
- mode: Sets the mode of the algorithm.

```
void init (float sample_rate, float *buf, int32_t npt, int32_t mode);
```

process

Processes the waveform to be generated, returning one sample. This should be called once per sample period.

```
float process (float &trig);
```

Mutators

set_amp

Sets the amplitude of the output signal.

Input range: 0-1?

```
inline void set_amp(float amp) { amp_ = amp; }
```

set_freq

Sets the frequency of the output signal in Hz.

Input range: Any positive value

```
inline void set_freq(float freq) { freq_ = freq; }
```

set_decay

Sets the time it takes for a triggered note to end in seconds.

Input range: Any positive value

```
inline void set_decay(float decay) { decay_ = decay; }
```

set_mode

Sets the mode of the algorithm.

```
inline void set_mode(int32_t mode) { mode_ = mode; }
```

Accessors

get_amp

Returns the current value for amp.

```
inline float get_amp() { return amp_; }
```

get_freq

Returns the current value for freq.

```
inline float get_freq() { return freq_; }
```

get_decay

Returns the current value for decay.

```
inline float get_decay() { return decay_; }
```

get_mode

Returns the current value for mode.

```
inline int32_t get_mode() { return mode_; }
```

port

Applies portamento to an input signal. At each new step value, the input is low-pass filtered to move towards that value at a rate determined by ihtim. ihtim is the "half-time" of the function (in seconds), during which the curve will traverse half the distance towards the new value, then half as much again, etc., theoretically never reaching its asymptote.

This code has been ported from Soundpipe to DaisySP by Paul Batchelor.

The Soundpipe module was extracted from the Csound opcode "portk".

Original Author(s): Robbin Whittle, John ffitch

Year: 1995, 1998

Location: Opcodes/biquad.c

init

Initializes port module

Arguments:

- sample_rate: sample rate of audio engine
- htime: half-time of the function, in seconds.

```
void init(float sample_rate, float htime);
```

process

Applies portamento to input signal and returns processed signal.

```
float process(float in);
```

Setters

set_htime

Sets htime

```
inline void set_htime(float htime) { htime_ = htime; }
```

Getters

get_htime

returns current value of htime

```
inline float get_htime() { return htime_; }
```

reverbsc

`Stereo Reverb`

Ported from soundpipe

example:

daisysp/modules/examples/ex_reverbsc

init

Initializes the reverb module, and sets the samplerate at which the process function will be called.

```
int init(float samplerate);
```

process

process the input through the reverb, and updates values of out1, and out2 with the new processed signal.

```
int process(const float &in1, const float &in2, float *out1, float *out2);
```

set_feedabck

controls the reverb time. reverb tail becomes infinite when set to 1.0

range: 0.0 to 1.0

```
inline void set_feedback(const float &fb) { _feedback = fb; }
```

set_lpfreq

controls the internal dampening filter's cutoff frequency.

range: 0.0 to samplerate / 2

```
inline void set_lpfreq(const float &freq) { _lpfreq = freq; }
```

svf

```
Double Sampled, Stable State Variable Filter
```

Credit to Andrew Simper from musicdsp.org

This is his "State Variable Filter (Double Sampled, Stable)"

Additional thanks to Laurent de Soras for stability limit, and Stefan Diedrichsen for the correct notch output

Ported by: Stephen Hensley

example: daisysp/examples/svf/

init

Initializes the filter

float samplerate - sample rate of the audio engine being run, and the frequency that the process function will be called.

```
void init(float samplerate);
```

process

Process the input signal, updating all of the outputs.

```
void process(float in);
```

Setters

set_freq

sets the frequency of the cutoff frequency.

f must be between 0.0 and samplerate / 2

```
void set_freq(float f);
```

set_res

sets the resonance of the filter.

Must be between 0.0 and 1.0 to ensure stability.

```
void set_res(float r);
```

set_drive

sets the drive of the filter, affecting the response of the resonance of the filter..

```
inline void set_drive(float d) { _drive = d; }
```

Filter Outputs

Lowpass Filter

```
inline float low() { return _out_low; }
```

Highpass Filter

```
inline float high() { return _out_high; }
```

Bandpass Filter

```
inline float band() { return _out_band; }
```

Notch Filter

```
inline float notch() { return _out_notch; }
```

Peak Filter

```
inline float peak() { return _out_peak; }
```