Psycle LUA scripting plugin API

1. Introduction

Psycle supports a variety of different plugins, providing you with additional sound effects and generators. One of these, The Lua Script Extension, empowers any user to develop a simple machine (or complex, depending on cpu requirements), simply with the help of a text editor (like notepad++), knowledge of the Lua scripting language, and this new API.
No compiler is needed, and changes to the code can be done live (See the "Reload script" option in the View menu of the parameters window).

This API is composed of three parts: The description of the skeleton for a working lua plugin, the description of the different Psycle modules that provide useful functionalities, and a working example in form of a step-by-step tutorial including a section on how to debug.

If you are new to Lua, a good starting point will be http://www.lua.org/.

2. Lua plugin skeleton

2.1 The Machine module and callback settings

Psycle is written in the C++ programming language and is unable to understand Lua scripts on its own. So, to be able to execute them, the application is linked to the Lua (currently 5.2.3) runtime environment, which provides Psycle with a Lua virtual machine. Don't worry, no need for esoteric readmes, you do not need to install anything apart from Psycle, the lua vm is now a standard distribution for all new psycle versions.
Once Psycle is able to execute Lua plugins, the next step is enabling it to communicate with them, which is what this API describes. Psycle needs to call methods provided by the Lua script (e.g. "work", which will process the samples) and "Luas" also need information from the Host, which they can obtain by calling methods offered by the C++ code in Psycle. The export and import of these functions is done via an instance of a psycle.machine module that will be registered to the host through the global function psycle.setmachine(mac). By overriding predefined callback methods, you fill the plugin with specialized code.
Finally, Psycle scans the LuaScript directory for lua files and checks if there's a psycle.machine registered and asks for the info method. You can create a subdirectory named after the lua file, and use this subdirectory to include multiple files within your plugin.

A good way is to only register the plugin in the main lua file and leave all other files as separate modules in the subdir. Note, that for all methods the psycle.machine module uses the ":" syntax and allows inheritance. (New operator has the ":" syntax, too.)

The import (host-callback) functions are

- info
- init
- work
- seqtick
- sequencetick
- stop
- command
- ontweaked

Example of registration

file: luadir/newsynth/machine.lua

```
-- create machine module

machine = require("psycle.machine"):new()
-- place for  specialized code
return machine
```

file: luadir/newsynth.lua

```
machine = require("newsynth.machine")
psycle.setmachine(machine)
```

To get started with the registration process you can also take a look at the examples we provided with the release.

2.1.1 Machine registration

```
info()
```

This called when the host checks for new plugins. The method should return meta data about the plugin. Here's what the return table should contain:

vendor	:	A string containing the author of the plugin
name	:	The plugin name that appears in the new machine dialog
generator	:	1 : is generator 0 : is an effect

version : version number
api : api number

Example :

```
function machine:info()
  return { vendor = ”James Synth”,
           name = “synthfingers”,
           generator = 1
           version = 007,
           api = 0 }
end
```

2.1.2 Machine initialization

```
init()
```

This method is called once when the Lua script is loaded by the host. It’s used by the plugin to initialize and create its variables and objects, additionally, you can set parameters (more under 2.2) that will be shown inside the Parameter Window.

Example:

```
filter = require("psycle.dsp.filter")
param = require(“psycle.parameter”)

function machine:init()
   filter1 = filter:new(filter.LOWPASS)
   p = require(“orderedtable”).new()
   p.fres = param:newknob("Filter Resonance", "", 0, 127, 127, 0):addlistener(self)
   p.fcut = param:newknob("Filter CutOff", "", 0, 127, 127, 127):addlistener(self)
   self:addparameters(p)
   self:setnumcols(2)
end
```

2.1.3 Parameter methods

```
addparameters(orderedtable)
```

It adds a *orderedtable* (see /luadir/psycle/orderedtable.lua) of parameters to the host.
We used this structure from the lua wiki, because the lua table structure itself is not ordered. The “orderdtable” keeps the order, and the varname (e.g. “lb1”) is used as an internal id. Now, if the user saves a psycle song, this id is used to find right parameter. This enables the plugin writer to later insert or change the order of the knobs without disrupting old songs.

Example:

```
param = require(“psycle.parameter”)
function machine:init()
  p = require(“orderedtable”).new()
  p.lb1 = param:newlabel(“lb1”)
  p.lb2 = param:newlabel(“lb2”)
  self:addparameters(p)
end
```

`setnumcols(cols)`

Sets the number of knob columns displayed in the parameter window.

Example:
(see 2.1.2 Init Example)

2.1.4 Sound processing methods

`work(num)`

This called to generate or manipulate the audio data the soundcard outputs.

`channel(index)`

This method is used inside the work method and returns the audio buffer. It serves as input and output buffer.

`numchannels()`

This method returns the number of channels available.

`numchannels(num)`

Sets the number of channels. It’s used inside the init method. If this method is not invoked, the

default number of channels is two.

Example:

```
filter = require("psycle.dsp.filter")

function machine:init()
   filter1 = filter:new(filter.LOWPASS)
   filter1:setcutoff(50)
   filter2 = filter:new(filter.LOWPASS)
   filter2:setcutoff(80)
   self:setnumchannels(2) -- actually this line is not needed
end

function machine:work()
   filter1:work(self:channel(0))
   filter2:work(self:channel(1))
end
```

2.1.5 Sequencer methods

```
machine:seqtick(channel, note, ins, cmd, val)
```

The SeqTick function is where your notes and pattern command handlers should be processed. Here's what the parameters mean:

channel
: the track number, where the event occurred

note
- (0-119) = Note on
- 120 = Note off

ins
: instrument

cmd
: command

val
: val

3. Psycle Lua modules

Psycle provides many functions by itself to be called from Lua scripts, like an envelope, oscillator generation, filtering, or more complex ones like generating and playing samples, or to chain and run other native and VST plugins from within the Lua script. These functions are provided in several independent modules which can be called as required by the script.

The available modules are:

- psycle.array
- psycle.parameters
- psycle.envelope
- psycle.dsp.math
- psycle.filter
- psycle.osc
- psycle.machine
- psycle.resampler
- psycle.wavedata

3.1 The psycle.array module

To work in an efficient way with sampled data, we've added the psycle.array module. A problem with embedding lua is that breaking the borders between c++ and the script by importing and exporting functions is quite expensive. Often plugin writers use loops to fill the buffer sample by sample. (e.g. a[i] = 2.0). Unfortunately this means that lua plugins invoke a set method to the host for each sample . Although the psycle.array module is able to have indexed acces, too, many things could be optimized by doing operations in a block operating a sequence of samples in c. These sequences can then calculated very fast in c++, and the number of expensive function calls is minimized. Note, that the psycle.array module uses as array index 0..N-1 (not the lua array indexing [1..N]) because this way is more common to sound programmers. You can pass lua arrays to the array constructor nevertheless, but be aware of the different indexing.

To create a psycle.array use the method *new*. Note that the new/arange method use the "." syntax.

new()

will generate an empty array. (internally even an empty array has a capacity of 256 double values to avoid memory resizings.)

new(num)

will generate an array with num elements and default value zero.

new(num, value)

will generate an array with num elements and filled with value.

arange(start, end, step)
generates an array with values in the interval [start, stop[(not including). The parameter sets the space between two values.

Further to the array methods described. The following methods use the ":" syntax.

fillzero()

This method clears the array values to zero.

copy(src)

This method copies the src psycle.array

copy(pos, src)

This method copies the src array right shifted by pos

add(number)

adds number to all array values

subs(number)

subtracts number from all array values

mul(factor)

multiplies all array values with factor

div(factor)

divides all array values with factor

rsum(value)

computes the running sum starting with value

resize(num)

This method changes the size of the array to num.

Additionally, the psycle array offers a set of non object related methods that accepts a psycle array as input and will return a newly constructed array. Instead of using the ":" you would use the "." syntax

sin(psycle.array)
cos(psycle.array)
tan(psycle.array)
pow(psycle.array)
sqrt(psycle.array)

Examples:

a = array.new(256)

3.2 The psycle.parameter module

require("psycle.parameter")

The parameter module is a native lua module that let you define three different kinds, knobs, labels and empty rows, that will be shown inside the parameter window. Internally it stores one double value in the normalized range from 0..1. The module offers methods to recalculate and work with the real value. The definition of the table you find in /psycle/parameter.lua. Additional the parameter module implements the observer design pattern. If the knob is tweak automatically listener are notified. To register the knobs to the host, you store them in a orderedtable, and use the addparameters method of the machine module.

To use it, first you need to require the module, e.g with param as table name:

param = require("psycle.parameter")

There are several constructors:

- newknob(n, l, m, ma, s, v)

- newlabel(n)

- newspace()

(used by the other create methods

- new(name, label, min, max, step, value, mpf))

Here's what the parameters mean:

name = name of the knob
label = label of the value
min = minimum range of the knob (double)
max = maximum range of the knob (double)
step = steps between minimum and maximum (integer)
v = startvalue of the knob (double)
(mpf = enum param.MPFNULL param.MPFLABEL param.MPFSTATE)

Often it's nesessarily, not to output the value, but to display a string literal, e.g if you want to select a wave form type like sin or saw. For that you can overwrite the display method, that as default returns the real value as string.

parameter:display()

Is called by the host to get a string representation of the value. Overwrite it to display your own format.

parameter:val()

This method returns the recalculated real value

parameter:setval(val)

This method stores a real value. Internally the value will be normed to 0..1

parameter:norm()

This method returns the normailized stored value witin the range of 0 ..1

parameter:addlistener(listener)

It's used to add a listener, that will be notified, if a value is tweaked. The listener needs to implement a method ontweaked(param), that will be invoked, if changes occur.
(You don't need to take care of removing this listener, because it's implemented as weak table. if no other reference to the listener other than that in the parameter module exists, the listener can be garbage collected.)

Example: /LuaScript/parameterdemo/plugin.lua

```
-- file : /LuaScript/parameterdemo/plugin.lua
local plugin = require("psycle.machine"):new()
param = require("psycle.parameter")
osc = require("psycle.osc")
shapes = {"sin", "saw", "square", "tri"}

function plugin:info()
  return  {vendor="psycle", name="paramdemo", generator=1, version=0, api=0}
end

function plugin:init()
    p = require("orderedtable"):new()
    p.shape = param:newknob("Waveform", "", 1, 4, 3, 1):addlistener(self)
    p.shape.display = function(self)
                      return shapes[self:val()]
                  end
    self:addparameters(p)
    osc1 = osc:new(p.shape:val(), 440)
    osc1:start(0)
end

function plugin:work()
  osc1:work(self:channel(0))
  self:channel(1):copy(self:channel(0))
end

function plugin:ontweaked(param)
```

```
  if param==p.shape then
     osc1:setshape(param:val())
  end
end

return plugin
```

```
-- file : /LuaScript/parameterdemo.lua
plugin = require("parameterdemo.plugin")
psycle.setmachine(plugin)
```

3.3 The psycle.osc module

require("psycle.osc")

The psycle osc module provides a wave oscillator, that can produce bandlimited sin, saw, square and triangle waves. Each shape has multiple waves for each octave to adjust the harmonics to fit the nyquist theorem and keep bandlimited.

To use it, first you need to require the module, e.g with osc as table name:

osc = require("psycle.osc")

Construction is done with the new method

new(shape, f)

shape = osc.SIN , osc.SAW, osc.SQR, osc.TRI
f = frequency in hz

start(phase)

Starts the oscillator with a given phase. Phase is in the range from 0..1.

stop(phase)

Stops the oscillator at the given phase

work(samples, [fm, [env])

Here's what the parameters mean:
samples = output to sample array. Wave data is added to existing values.
optional:
fm = frequence modulation array (added each sample to the base frequency)
optional:
env = amplitude envelope, multiplied to each output sample

This method returns the number of processed samples

setfrequency(f)

frequency()

setgain(g)
sets the maximum amplitude
gain
This method returns the maximum amplitude

isplaying()

Checks if the oscillator is playing.

Example:

```
osc = require("psycle.osc")
array = require("psycle.array")

function plugin:info()
  return  {vendor="psycle", name="oscdemo", generator=1, version=0, api=0}
end

function plugin:init()
    vibrato = osc:new(osc.SIN, 3)
    vibrato:setgain(5)
    vibrato:start(0)
    fm = array.new(256)
    osc1 = osc:new(osc.SAW, 440)
    osc1:start(0)
    volosc = osc:new(osc.TRI, 0.1)
```

```
    volosc:start(0)
    am = array.new(256)
end

function plugin:work(num)
  fm:resize(num)
  fm:fillzero()
  am:resize(num)
  am:fillzero()
  vibrato:work(fm)
  volosc:work(am)
  osc1:work(self:channel(0), fm, am)
  self:channel(1):copy(self:channel(0))
end
```

3.4 The psycle.resampler module

require("psycle.resampler")

The psycle resampler module is able to produce resampled data. The module uses internally code from sampulse and offers its interpolation methods, currently zerohold, linear and sinc.
The resampler needs to be created with a wavetable containing a frequency range and a wavedata. A wavedata contains the data to resample and additional meta information, like loop points, wavetune and wave samplerate.

The creation method is :
new({wave1, lo, hi }[, {wave2, lo, hi}, …])

You need to start the resampler with

start([phase])

The optional parameter phase is a normalized value 0..1 of the loop(or wavelength) length setting the sample position. If not set, the default is 0.

The method

isplaying()

returns a boolean value. True means, the resampler is playing, false means it has stopped.

The method
stop(phase)
stops the current playing with the given phase.

The method
noteoff()
releases a looped wave from the sustain part.

The method
work(left, right, fm, env)
generates the resampled data.
Here's what the parameters means:
left = output will be added to the left psycle.array or set nil
right = output will be added to the right psycle.array or set nil
fm = frequence modulation psycle.array added sample by sample to the base frequency at playing
amp = amplitude psycle.array multiplied to the resampled data output sample by sample

3.5 Loading other libraries

You can load all plugins that Psycle has registered. A list of available plugins can be seen from the New Machine Dialog in the Psycle host. To load other plugins load the psycle.machine module, and call the new() method with the name of the plugin (.dll file name):

You create and load a plugin with:
machine = require("psycle.machine"):new(plugin).

For all other methods you can refer to the Section 2.1

3.6 Using existing machines in the machineview

You can access existing machines in the machienview with the module psycle.machine, too
To obtain an existing machine use:

machine = require("psycle.machine"):new(macindex)

Macindex can have values between (0..255). If no machine exists, new returns nil.

For all other methods you can refer to the Section 2.1

3.7 The psycle.envelope module

require("psycle.envelope")

The psycle envelope allows you to change the volume, the frequency or the filter cutoff during the reproduction of a note. You can do so in the methods of other modules, like the resampler or the waveosc, which accept psycle.array parameters to modify the sound.

To create an envelope require the psycle.envelope module. You can create an ahdsr with:

newahdsr(attacktime, holdtime, decaytime, sustainlevel, releasetime)

or you can define your own stages. This gives you complete flexibility, but the creation is more complex:

new({{time1, peak1}, {time2, peak2}, .. }, sustainpos [, startvalue])

Here's what each parameter means:

time : The duration of one stage in seconds.
peak: The ending position of a stage.
sustainpos: The index (counted from 1) of the place where it holds while it is sustained.
startvalue: An optional value to indicate the initial index where it starts to play. Default value is 0, which means start from the beginning..

After creation you need to start the envelope with:
start()

The envelope is in playing mode. You can check the mode with

isplaying().

This method returns true, if there stages need to be worked, or false, if all stages are finished ending with the peak value from the last stage.

work(num)

This method returns a psycle.array of num samples containing the current envelope data.

release()

After invoking this method, the envelope will play from the sustainpos to the last stage and finally isplaying() will return false when the envelope processes the last stage.
If the sutainpos isn't reached yet, the envelope will jump to the release stage. The release stage begins with the last reached value from the previous stage.

if you need to change the envelope times itself you can call:

settime(stage, newtime)

or if you want to change the peak you can call:

setpeak(stage, time).

You can also call

time(stage)

and

peak(time)

to get the current settings of the stage.

The method

stage()

returns the current stage that the envelope is processing.

3.8 The psycle.filter module

require(“psycle.dsp.filter”)

3.9 The psycle.dsp.math module

4 Writing a lua plugin

This chapter is going to explain how to make a simple Lua plugin. It is written as a descriptive tutorial on how to make your first plugin and see the fundamentals of the API working.

4.1 The header of the Lua plugin

Now, it's time to write your first Lua synth plugin. Open your favorite texteditor (e.g. notepad++ or zerobranestudio) and write/copy the following text:

```
machine = require(“psycle.machine”):new()
-- plugin info
function machine:info()
  return
      {vendor="superman",
       name="supersynth",
       generator=1,
       version=0,
       api=0}
end

return machine
```

Save this code as machine.lua in a new subdir inside the Lua scripts directory like:

LuaScript/supersynth/.

Now create a second file supersynth.lua, place it in the LuaScript directory and copy the following code into it:

```
machine = require("supersynth.machine")
psycle.setmachine(machine)
```

Run Psycle. Now check if the new plugin appears. To do this, press F9, or double click on the Machine View and the new machine dialog appears. Now click the Button labeled "Check for new plugins". In the left treeview "supersynth" should appear (either in the Generators branch, or the Lua Scripts branch, depending on how you setup the tree view). In that case, congratulations, you just wrote your first Psycle Lua plugin. You can change the strings "superman" and "supersynth" to change the author name, and the plugin name that will be shown inside Psycle.

Now, let's move on to learn how to make some sounds.

4.2 Producing sounds

4.2.1 What is sound, and how does a computer reproduce it

Any sound you hear are waves oscillating through the air. Such waves can be represented graphically by an oscilloscope (see figure 1)

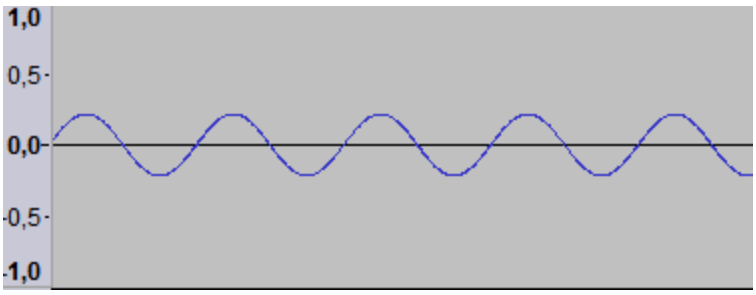


Figure1
A sinus curve

In the nature, the time resolution is continuous and has endless points within a time interval. On the other hand, a computer can only store a finite number of samples, so when they are stored and transmitted in the digital domain, the signals are discrete in time (and amplitude). Each one of these discrete values is called a sample. The sample rate indicates, how many of these samples are required for exactly one second of sound. Figure 2 shows a part of the wave from figure with the discrete points marked.

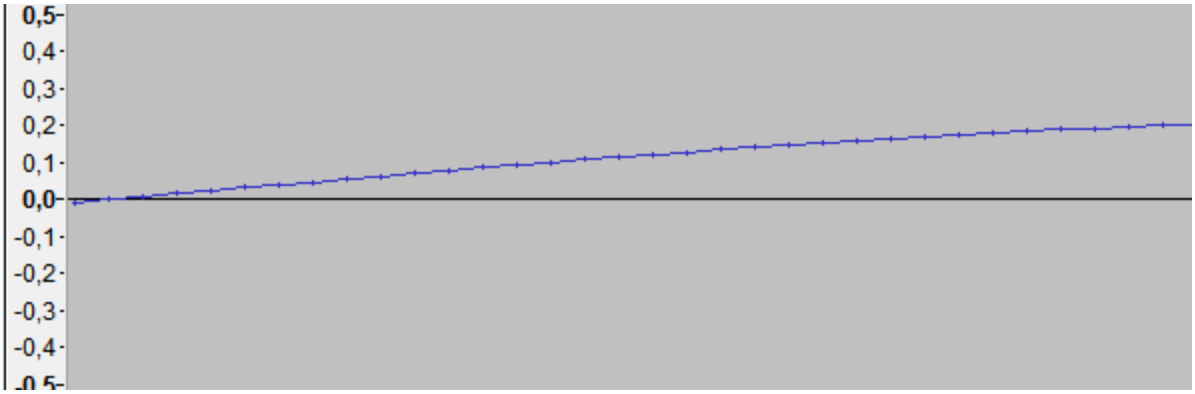


Figure 2
A sinus curve with discrete values

To reproduce it, your soundcard has a sample buffer and obtains a constant number of samples from there, in a small time interval (depending on the sample rate). These samples are passed to the DAC and its reconstruction filter. These two elements of the soundcard are able to convert the signal to a continuous sound again, which will be played through your speakers or headphones.

You are invited to check articles on wikipedia or other places if you want to learn more about how soundcards are able to do this transformation, and why increasing the sample rate and bitrate does not translate in a better signal if the conditions of the Shannon theorem are accomplished and the limits of the human hearing are taken into account.

4.2.2 How to instruct our lua plugin to generate sound

Psycle has a method called Work which is repeatedly called to fill the sound buffer with a specific number of samples. In this step of the example, we will use the module psycle.osc, which can produce four wave shapes: sinus, sawtooth, square and triangle. We will use this module to output a saw to Psycle's Work buffer, which will be played by the sound card.
The first step is to create an osc. We do so in the init callback method. This method is called once when the Lua script is loaded by the host. The require statement loads the psycle.osc module. Then an instance osc1 is created with a sawtooth wave shape and a frequency of 440 Hertz.
Finally, in the work method, we instruct osc1 to output its sound to Psycle's channel 0.

Open again the supersynth.lua script (if you closed it), and append the following code:

```
-- include our osc module
osc = require("psycle.osc")
-- plugin constructor
function machine:init()
```

```
  osc1 = osc:new(osc:SAW, 440)
  osc1:start()
end

function machine:work(num)
  if (osc1:isplaying() then
    osc1:work(pc:channel(0))
  end
end
```

Please, check your soundcard volume at this step, we don't want you to get a loud sound out of your speakers.
If you still have the plugin loaded in Psycle, double click on its machine in the machine view (or press Shift+Enter). An empty Parameter View will open from where you can use the View->Reload Script menu option, or press Alt+R to reload the script. Now your plugin should be reloaded and Psycle should be playing saw wave. Mute the machine or delete it to make the sound stop.
Whenever you make changes to your code, you can reload it in this way.

4.3 Events

4.3.1 Noteon and NoteOffs

A Psycle generator does not play a sound signal the whole time. Instead, it reacts to NoteOn and NoteOffs events produced by the sequencer or a keystroke by the user. For that, the host calls for each of such an event the seqtick event method. Here's how to override it:

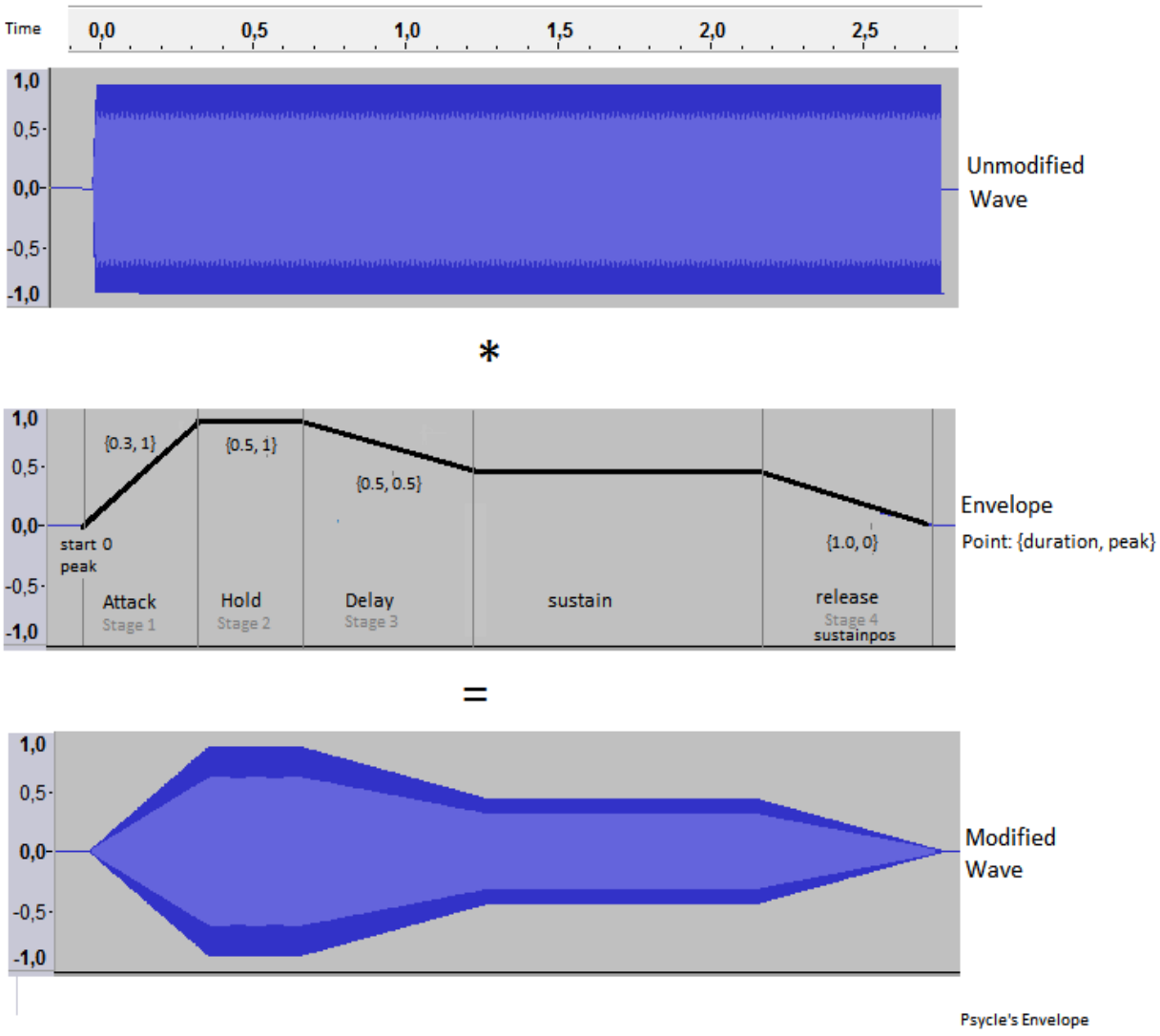
```
local dspmath = require("psycle.dsp.math")

function machine:seqtick(channel, note, ins, cmd, val)
   if note<120 then   -- noteon
      local f = dspmath.notetofreq(note)
      osc1:setfrequency()
      osc1:start()
   elseif note == 120 -- noteoff
      osc1:stop()
   end
end
```

The parameter note tells us, if a noteon or a noteoff occurred. Values below 120 represent the midi key value, the pitch of the noteon in semitones.
To get the pitch frequency from a midi keyvalue there's a conversion method in the psycle.dsp.math module.
osc1:start() starts the osc and osc1:stop() stops it.

4.3 A Noteon off is much more

In the previous you learned how a Psycle Lua generator reacts on note on and offs. But the sound is still quite boring. To bring in more dynamics other Psycle synths use envelopes. Envelopes are used to modify the amplitude, frequency and filter behaviour.
Let's start with the amplitude. Real Instruments don't play a note at the same volume all the time, but change it gradually.
A string instrument starts more softly than a loud snare drum. To modify the sound in this way you can multiply your oscillator output by an envelope. The next Figure demonstrates this.



4.3 Polyphony with Voices

The above demo works well with monopohonic synths. But how can you play multiple notes with the same synth?

A good way to do this is to create a new module “voice”, that encapsulates all the sound generating process of a single note. The synth itself can then have multiple voices that play several notes in parallel . Usually a voice consists of one or more oscs and a volume envelope (sometimes called voltage controller), that controls the volume for the duration of a note.
On noteoff, instead of aborting the sound abruptly, the note will softly fade out. This stage is called release. So a volume envelope has a second important task, to determine if a voice should be playing or not. Additionally a voice can also have frequency modulation controllers, that add effects that control

the pitch of a note during playing. A voice bundles all these sound processing. Several instances of a voice are then able to play in parallel multiple channels.

A simple voice module could look like this:

```
local voice = {}

function voice:new()
  local v = {}
  setmetatable(v, self)
  self.__index = self
  v.vc = envelope.newahdsr(1, 1, 1, 0.5, 1)
  v.osc = osc:new(osc.SAW)
  v.osc:start()
  self.:isplaying = false
  return v
end

function voice:noteon(note)
    self:isplaying = true
    local f = dspmath.notetofreq(note)
    self.osc.setfrequency(f)
    self.env:start()
    self.env:settime(4, 0.5)
end

function voice:noteoff(note)
   self.env:release()
end

function voice:faststop()
  self.env:settime(4, 0.03)
  self.env:release();
end

function voice:work(samples)
   self.isplaying = self.vc:isplaying()
   osc:work(samples, nil, self:vc)
end
```

return v

Using the the common lua OO design pattern, the voice implements these methods:

- new
- noteon(note)
- noteoff()
- faststop()
- work.

With new you can create a instance with one saw oscillator and an AHDSR envelope
A boolean isplaying is used to control the playing state of the voice. The methods Noteon and Noteoff can handle the events from the host. faststop is called by the Host to stop all sounds from the synth. This implementation will decay the envelope fast, but avoiding crackles that could be caused if the sound stopped immediately.

Ok, now let's see how we can handle the new voice module in our machine.lua module.

Many synths have a hardcoded number of voices. This number is often 32. The advantage is that a voice doesn't need to be recreated during a noteoff event or deleted if a noteoff had occurred. Drawback is higher memory usage and if parameters are used, a higher effort to keep all the voices updated when knob values are changed.
By default there is a pool of free voices in an array. Because each voice has a isplaying flag, the machine host can select the ones that are in playing state, else they aren't working.
If a noteon is received, the synth can search the voice pool for a free one, (one that returns isplaying false) and lastly call the noteon method of this voice.
The same thing is done again whenever a new noteon is received.

Let's consider what we need to do if a noteoff occurs.
A noteoff calls seqtick with the cmd value of 120 and additionally a channel index. In Psycle, a noteon and noteoff belongs to a channel. A channel corresponds to a track in the pattern editor. All noteon and noteoff events that occur in one track/channel occur together and the synth needs to map this correctly. One solution is to store the used channels and the used voices. If a noteoff event occurs, we know then which voice should be muted.
If one voice is playing a note in channel A, and in the same channel comes a new note, we have two options to realize a crackle free transition. We could just update the frequency of the voice, but then our attack phase would be missing; or we make a very fast fade out of this voice and search a new free voice, that will start to play parallel with the new note. This second optioncan be achieved if we call

faststop on the current channel voice and we search a free voice in the voice array and invoke a noteon.

Here are these ideas implemented, except one detail, we start with 6 hardcoded voices, and if we need more, a missing voice is added to the voice array. Notice that we never delete a voice.

local voices = {}
local channels = {}

```
function machine:init()
  for i=1, 6 do voices[#voices+1]=voice:new()
  for i=0, 64 do channels[i]=1 end
  self.currvoice = 1
end

function machine:freevoice()
  local v = voices[self.currvoice];
  local c = 0
  while v.isplaying do
    if self.currvoice == #voices then
      self.currvoice = 1
    else
      self.currvoice = self.currvoice + 1
      break
    end
       c = c + 1
       if c > #voices then
          voices[#voices+1] = voice:new()
         self.currvoice = #voices
       end
  end
end
function machine:seqtick(channel, note, ins, cmd, val)
  local curr = voices[channels[channel]]
  if (note < 119) then
      if curr.isplaying then
```

```
        curr:faststop()
      end
      self:freevoice()
      channels[channel] = self.currvoice
      curr = voices[self.currvoice];
      curr:noteon(note)
    end
  elseif (note==120) then
     voices[channels[channel]]:noteoff()
  end
end
```

4.5 How to debug your plugin

Debugging your new plugin is possible with remote debuggers. You can find here a list of possible remote debuggers and IDE's:

http://lua-users.org/wiki/LuaIntegratedDevelopmentEnvironments

Psycle uses lua 5.2.3, the latest official framework release. Unfortunately, some debuggers work only up to lua 5.1 (e.g. decoda).

One that works with 5.2 is the Open Source Debugger and ide called ZerobraneStudio. Here is where you can obtain zerobrane: http://studio.zerobrane.com/

Here follows an explanation on how to use it:

First you have to add this line to the beginning of your script :

`require('mobdebug').start()`

Next, open Psycle and ZerobraneStudio and do this:

1. Project | Start Debugger Server
2. Open the machine.lua then select the menu : Project | Project Directory | Set From Current File.
3. Load your lua machine into Psycle's machine view

The Zerobrain application tab starts blinking. You can press play to continue execution.

If you wish to set a breakpoint, you first need to pause zerobrain, then double click left to set a breakpoint you wish. You can then step over step into with the buttons you see in the top toolbar.
You can see variable values, if you move the mouse over the declaration in the code.
You can also print to zerobrane's Output Window with the print statement. Unfortunately this doesn't work right out of the box. You need to add in Zerobrane's Console Window this line first:

```
DebuggerAttachDefault({redirect = "c"})
```

Then you can add in your script a print statement like
`print('i am before work:'..num)`
and the output appears in zerobrane.

For more documentation please refer to the zerobrane website. A good idea is to install one of the many Windows window enhancer (e.g. 4th tray minimizer), that adds to your window a stay always on top feature.
If you finished debugging, don't forget to remove or outcomment the `require('mobdebug').start()` statement.

Most of this steps are documented here. You don't need to care about the part about mobdebug.lua and path settings, Psycle has already prepared it for you.
http://studio.zerobrane.com/doc-remote-debugging.html