1

Pd Tutorials for LICH & WITCH Devels

1. Pd.Tutorial Essentials

Tutorial by Xavi Manzanares
http://xavimanzanares.oneshaptiques.space

LICH Module by ://
Befaco & Rebel Technologies

PD
FOR
LICH
&
WITCH

pd~

1. Pd.Tutorial Essentials

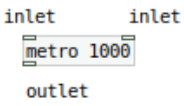
LICH requirements >
Pd Vanilla, in other words the original kernel of Pd without external libraries developed by the community
Downloads > https://puredata.info/downloads/pure-data

Note : In order to follow the next instructions, download this tutorial in .pd format
This pdf will be anyways useful to read whenever you don't have your computer to practice.

0. Before Programming

Before programming is useful to know some basic issues in Pd environment.
As you may know, Pd is an open sourced graphic programming language which controls the DSP in a dataflow of structures that you can build.

How this dataflow works?
Generally speaking, data flow works from top to bottom, and from Right to Left in the GUI.
So Programming elements may have inlets at the top of them and outlets at the bottom.



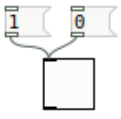
Left inlet activates a particular function [object] in this case 'metro' that is a metronome or clock function

We can both init this function with [bang] or [toggle] elements we find in Put Menu

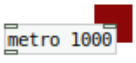
With [bang] once is started the object will be activated until we close the patch or programming GUI surface

With [toggle] we can start and stop the function [object] any moment we need

Binary Messages can be linked to a Toggle to activate it

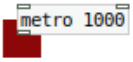


Right inlet changes the argument (or value) described in the function, in this case 1000



note that arguments or values does not have a specific units, but it directly applies to a specific object.

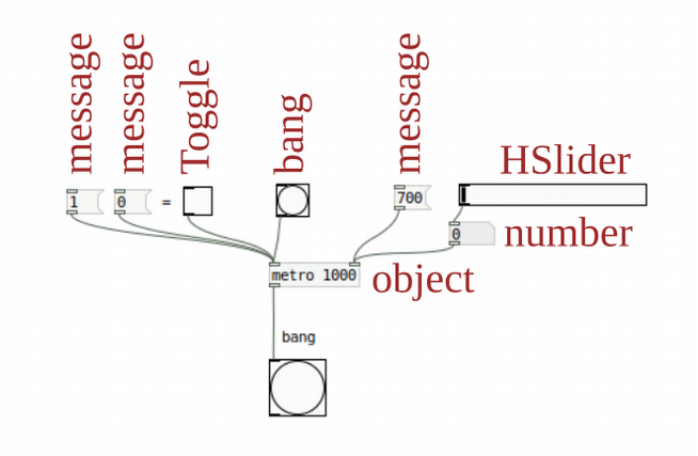
for example if we find metro 1000, the argument it means 1000 milliseconds. If the object is osc~ 1000, the argument means 1000 hertz



outlet in the case of [metro] is a trigger which in Pd is the element 'bang'

bang

Like other languages we can build structures with different methods and solutions.
In this example, there are different ways to activate the function metronome (object **[metro]**) :
Through messages **[1 [** and **[0 [**
Through a **Toggle** that in fact is an On / Off switch
Through a **bang** or a trigger



On the other side, on the right inlet we are introducing a new value for a particular argument.
This can be a fixed value with **message**, or a dynamic value with **HSlider** or **number**
Remember that even could be a default written value in the object (like 1000 in this example), the last value we are dynamically modificating in live (for example thorugh the associated Hslider or number), will be defined as the lead and 'definitive' one.

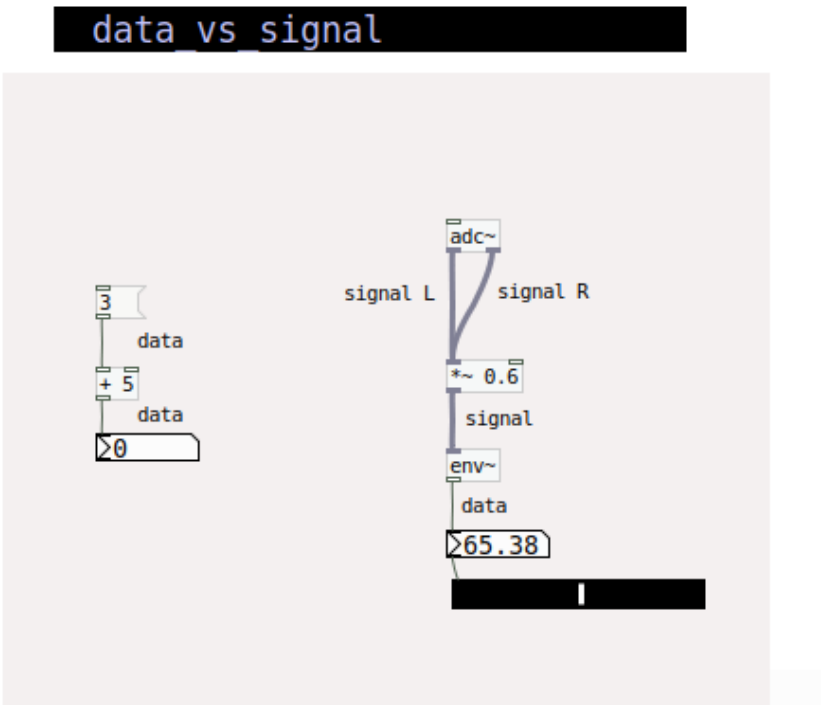
There is another issue important to take it into account before programming :
In Pd we have **data connections** and **signal connections.**

Data connections sends numbers and alfanumeric messages, at the speed of your CPU can manage and delay (latency) that has to be defined in your Pd preferences (depending on the OS can be found in the menu **Edit > preferences** or in the menu **Media > AudioSettings**).

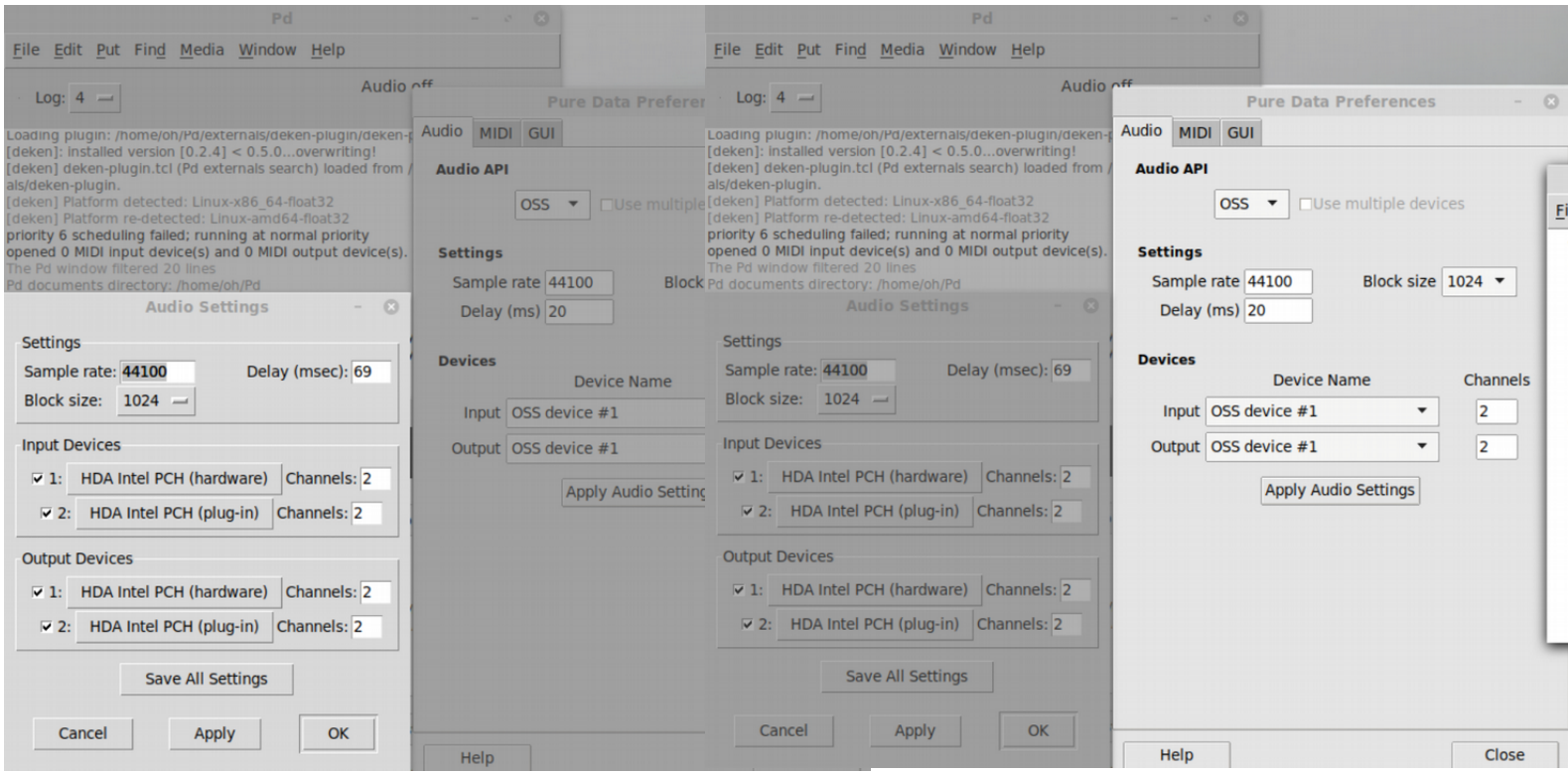
Signal connections sends / receive signals at 44100hz (so 44100 dots/samples every second) or whatever samplerate you already set up in the preferences of Pd.

Data** connections are **thin 'cables'
Signal** connections are **wide 'cables'
*Any object or message for **Data** connections **is written without tilde ~***
*Any object or message for **Signal** connections **is written with tilde ~***

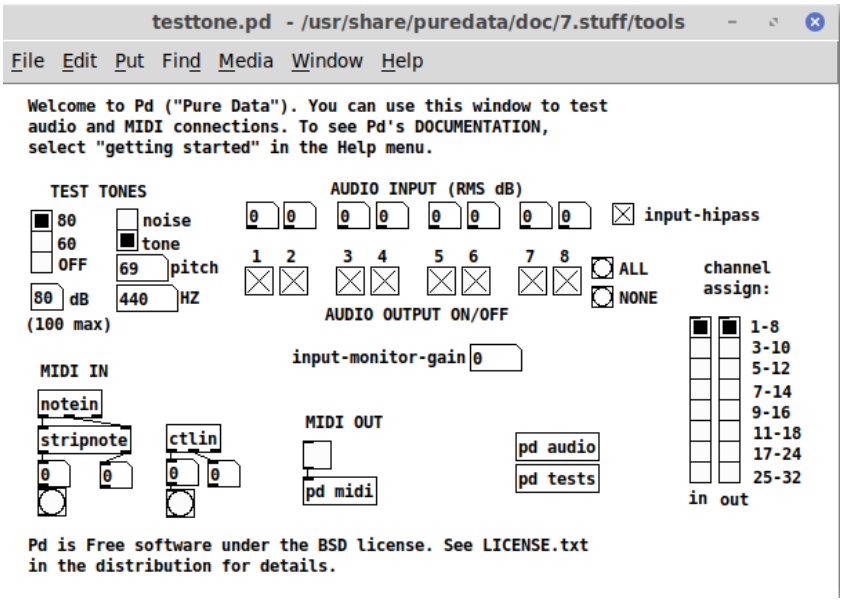
For example a multiplier of *data* is **[* 2]** and a multiplier of *signal* is **[*~ 2]**



In **preferences** you can manage different audio devices like internal /external soundcards or even working with several soundcards.
These GUI interfaces can change depending on the OS you're using.



In **preferences** also you can change **samplerate** or **latency (Delay).**
Latency (Delay) can produce errors for short values, depending on the size / calculus of your algorythms and the power of your computer processor. So it's a value that we can tweak in order to work DSP fine and without glitches. But how we know it?
The first and basic test to check if pd is working fine is **Test audio and Midi** in the menu Media :



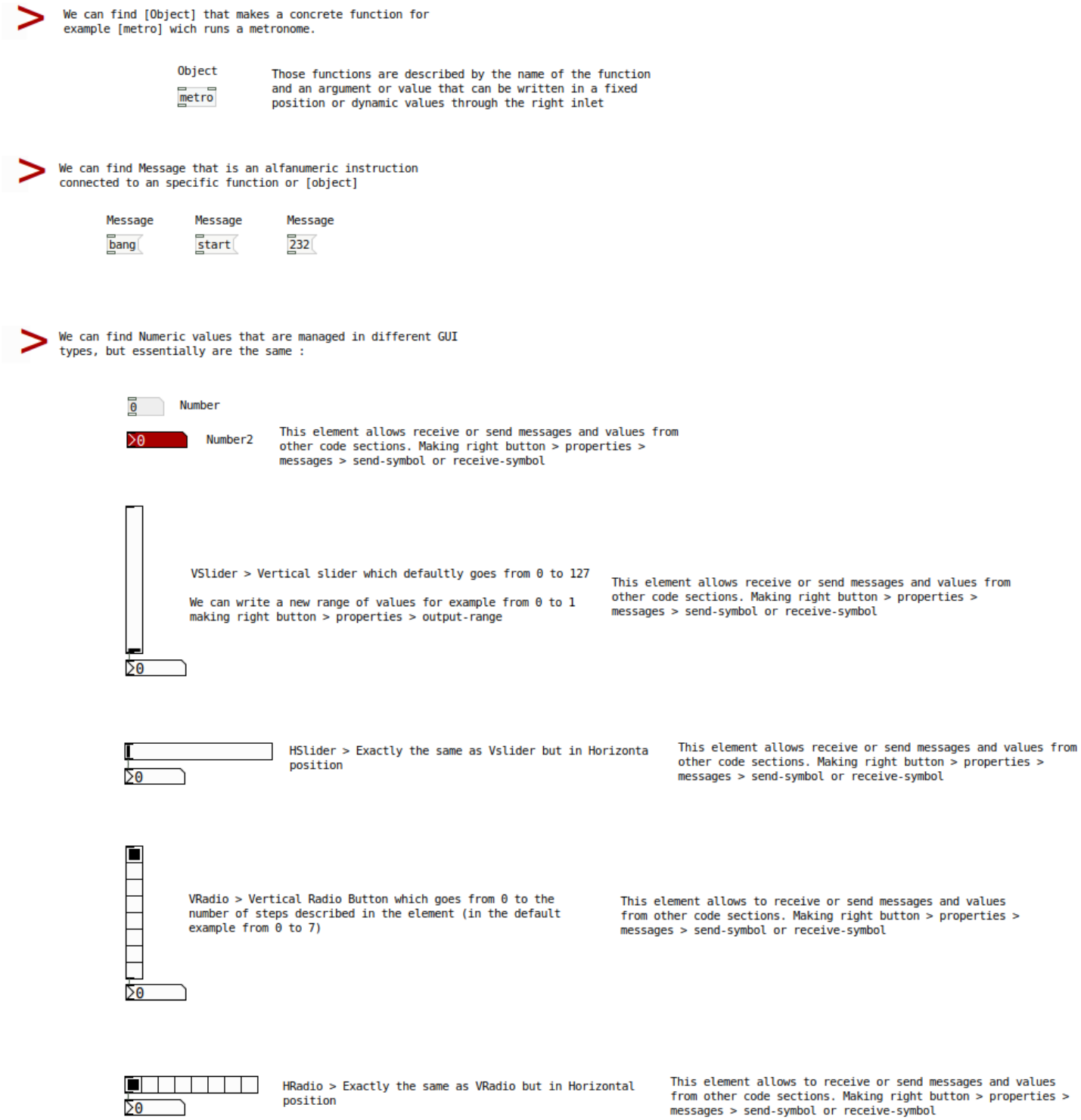
We can select Test Tones button on the left side, and select 80db.
<u>If pd is working fine a sine tone of 440hz will be output in your soundsystem or headphones.</u>

When we create a new file with Ctrl+n it appears a blank GUI surface where you can build your applications. This suface is called **Patch** like when we create a set of concrete connections in the Modular Synths Cosmos.

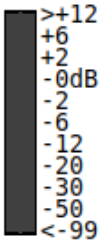
Finally, remember that Pd is a dataflow Programming environment where we can build DSP applications, but also manage and play with them as a performance/musician mode.
Maybe this is the command you'll be using the most if you program with Pd > **Ctrl+E**
Ctrl+E will switch between these two modes :
> Edit / Programming Mode
You can move and writte whatever you want in the patch
> Performance / Musician Mode
You can just only move the sliders and dynamic elements of the Code as a performer

syntax
1.1. Pd syntax

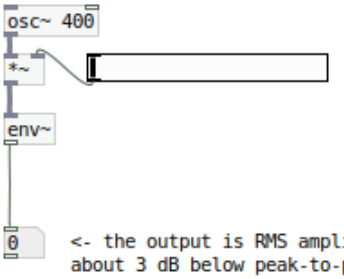
Even we can find several menus in the Pd shell or in any New patch from scratch (**File / Edit / Put / Windows / Media / Help**), there is one menu that is related to the programming language elements, and therefore the most important one, which is the menu **Put**.
In **Put** we'll find different elements of programming :



VU is a vumeter which can visualize dB from incoming RMS signal



VU is an element not enough straight forward to monitor amplitude values. Instead of this, you can check it with [env~] which features the envelope value



<- the output is RMS amplitude which (for a sinusoid) is about 3 dB below peak-to-peak amplitude.

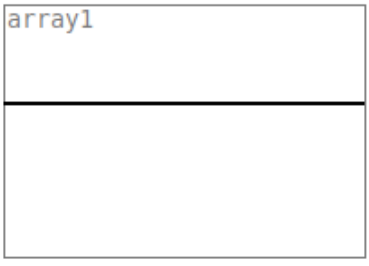
Canvas is a GUI element which features design zonification of the code.

This element allows to receive or send messages and values from other code sections. Making right button > properties > messages > send-symbol or receive-symbol

Graph allows to manage different code sections as a 'backend' swithcing 'graph on parent' after making right button > properties of this element.
It will be useful to save space in the main GUI

Array is a memory of numeric values, very useful for different algorythms we want to program.

This element will be explained later due to the fact of its versality and strenght.



syntax
1.2. Pd Programming Dataflow

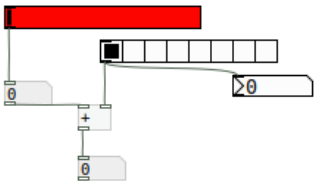
data.flow

PD Pure-data is a Datalow visual programming language

Flow is from up to bottom and from right to left

Is important to rearange different algorythms which can be automatized in the flow

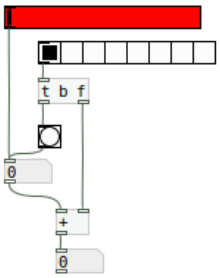
For example in this piece of code, any time we move HRadio button we need to 'refresh' the value of the red slider



if we want to automatize it, we can put the Red slider and the Hradio associated with the [t b f] object

[t b f] object describes 't' trigger 'b' bang and 'f' float

Therefore any time we move or the slider or the HRadio button the operation will be directly updated



sends&receives

Messages and values can be send and received anywere in a patch, building a rhizomatic structure with the only one hierarchy that have been constructed.

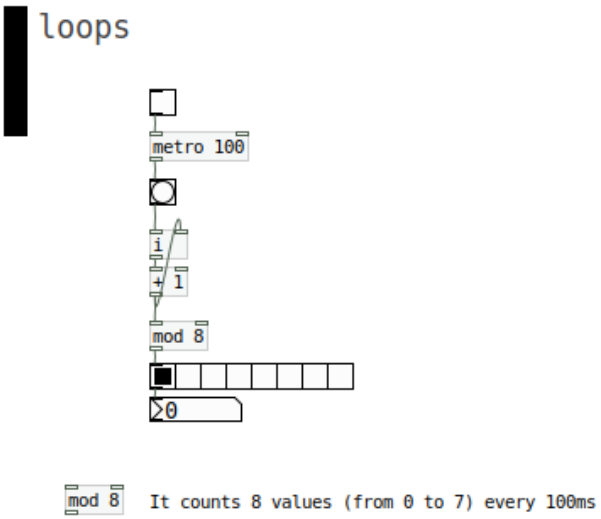
The rhizomatic structure of the data flow it has no hierarchy, except the order that any code element has been constructed.

Basic algorythms
2.1. Pd Basic Algoryhtms

In the next examples we'll see a bunch of tiny algorythms and sections of code which can be useful for many developments.

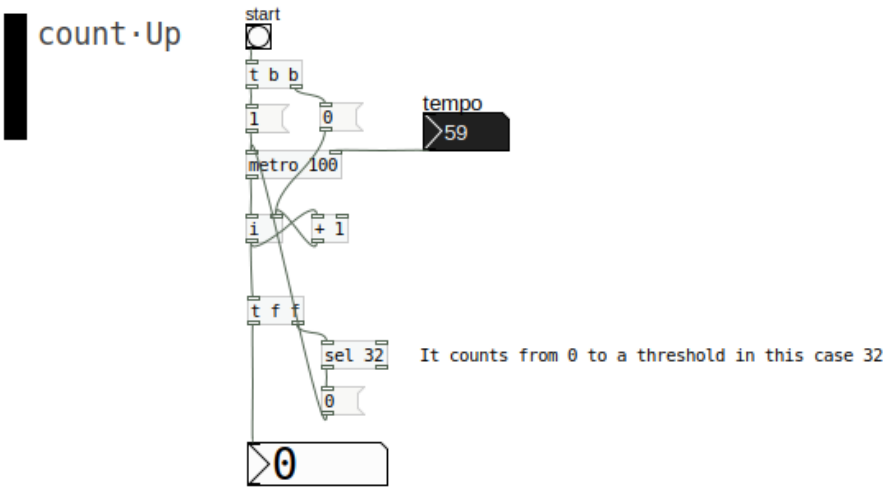
loops

Like other programming languages, one of the operations we are using to build structures in time are **loops**.
Even other programming languages this feature is essential for reading the whole code, in Pd we can make several groups of loops that are running apart from each other. Therefore we can build several groups of loops to make different operations at the same time.
Here there is an example of how to build a loop of 8 steps (from 0 to 7). In this case the steps are changing to a fixed speed of 100ms, driven by the object metro.

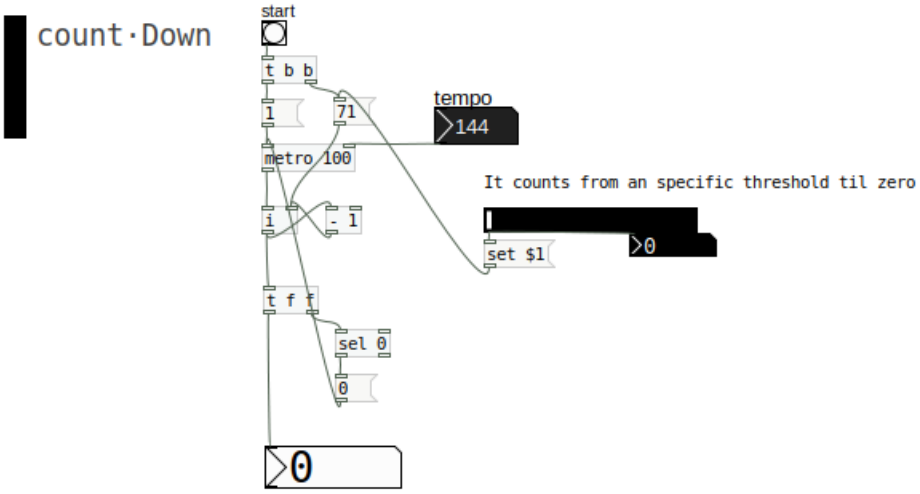


Counters

A counter builds a sequence of numbers until to a certain value.
If we want to make a ***count·up*** sequence we can use this piece of code:

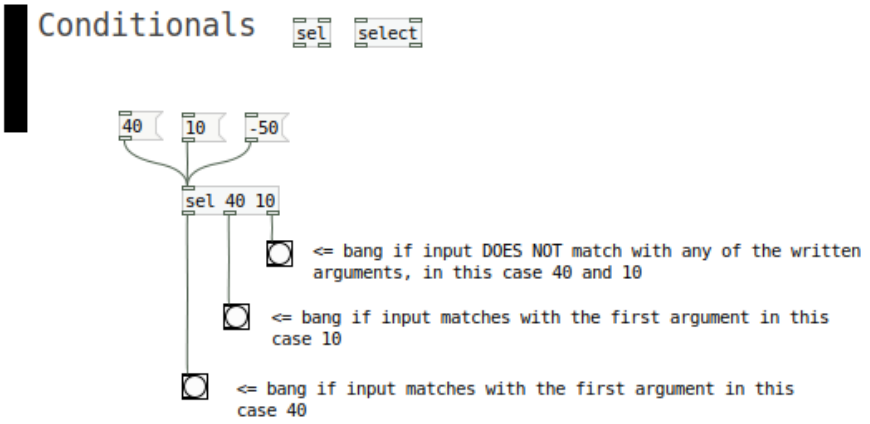


On the contrary, if we want to make a ***count·down*** sequence we can use this piece of code:



Conditionals

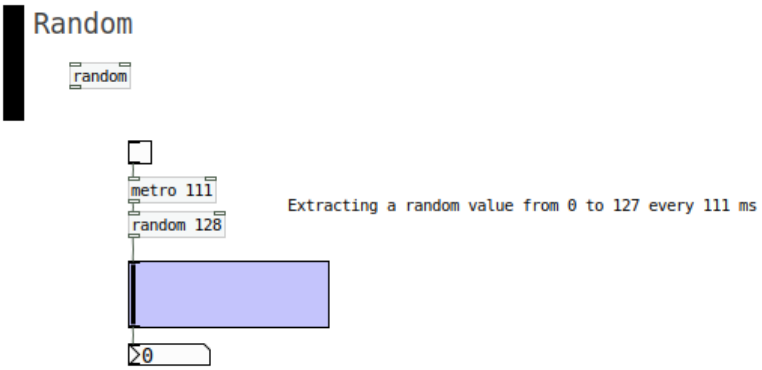
Conditionals are the classic if, else etc. functions from another languages.
In Pd we can solve conditional dataflow with the object **[sel]** or **[select]** which solves both the *if* condition and *else* condition.
The object **[sel]** works with a *string* of numbers or alfanumeric values, where each value has an specific outlet, ordered from left to right.
Notice that in this example we have 2 arguments (40 and 10), but anyways we have 3 outlets in the object . That's exactly the reason that the first outlets corresponds to an *if* function, and the right side outlet works like an *else* function.



There are other functions [objects] to manage conditional structures, for example :
[route] > distributor > similar to **[sel]** but a bit more complex and not so straight forward
[spigot] > gate / door > allow continuity or not from the incoming dataflow
[moses] > splitter > splits numeric values into two data lines from an specific argument ex. [moses 34]

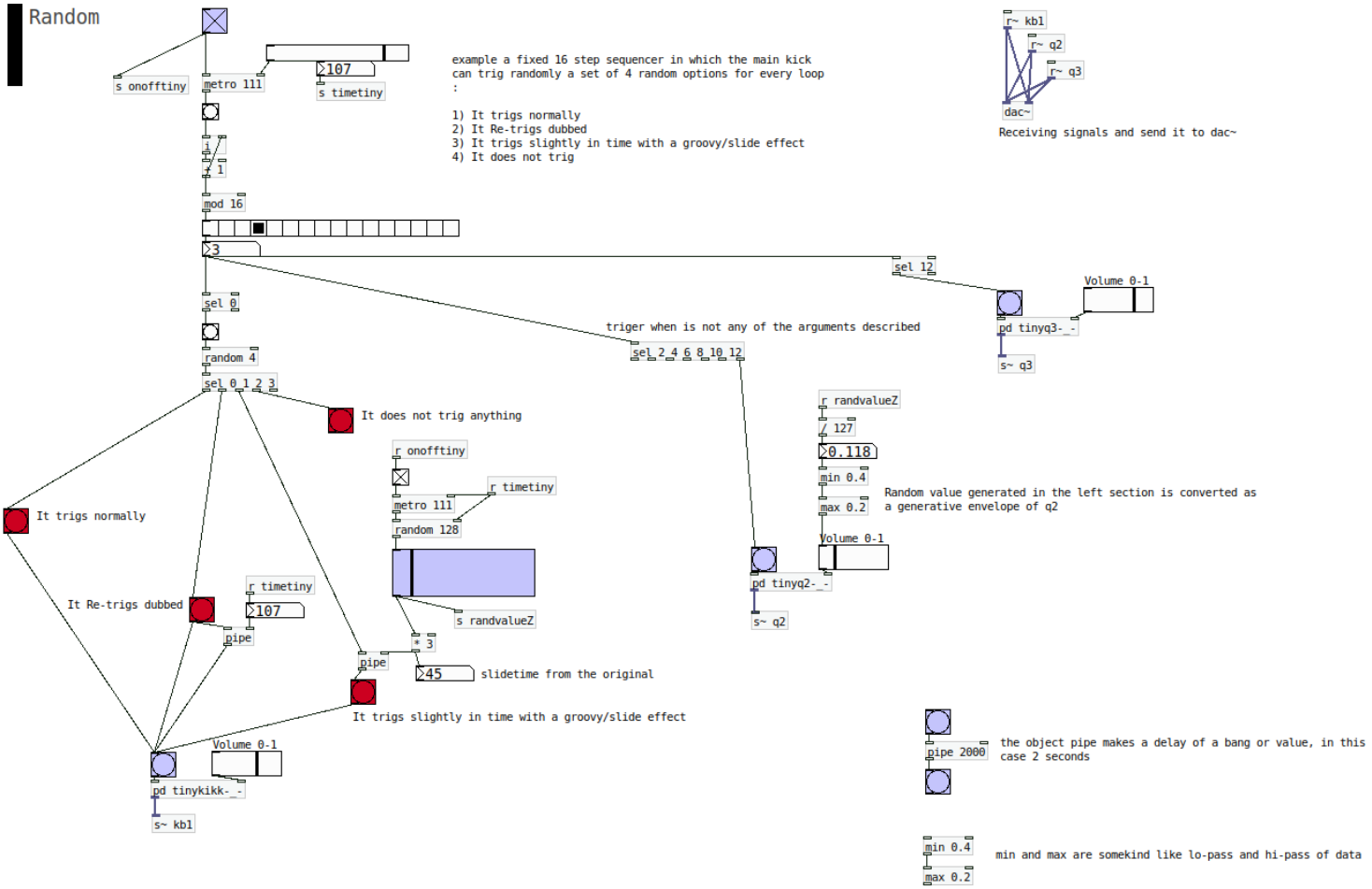
Random

Maybe one of the most exciting feature in Computer Science is the **random** function.
Of course is a very useful tool if you want to make ***generative instruments***, but remember that even we adore randomness, if we overuse this function, this could not solve our expectations in the sonic design.



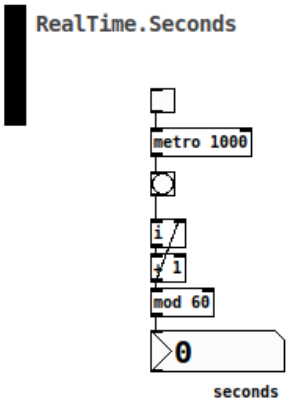
One suggestion is to use random methods in a very defined set of options.

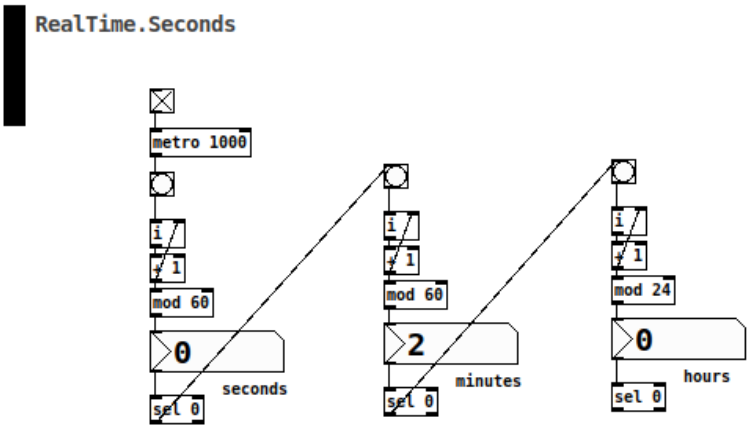
For example we can build a fixed 16 step sequencer in which the main kick can trig randomly a set of 4 random options for every loop :
1) It trigs normally
2) It Re-trigs dubbed
3) It trigs slightly in time with a groovy/slide effect
4) It does not trig
In other words, for this example using randomness in order to get more or less static/patterned structures and organicity/changes at the same time.



Real Time Clock

Maybe we need to use a real time clock (seconds, minutes etc) for generative instruments we want to make changes in time, for example an Installation in which we want to take it into account what time is it, or changing parameters depending if we are on the morning, afternoon etc.
With a set of conditionals **[sel]** we already seen we can trigger some events for specific daytime.

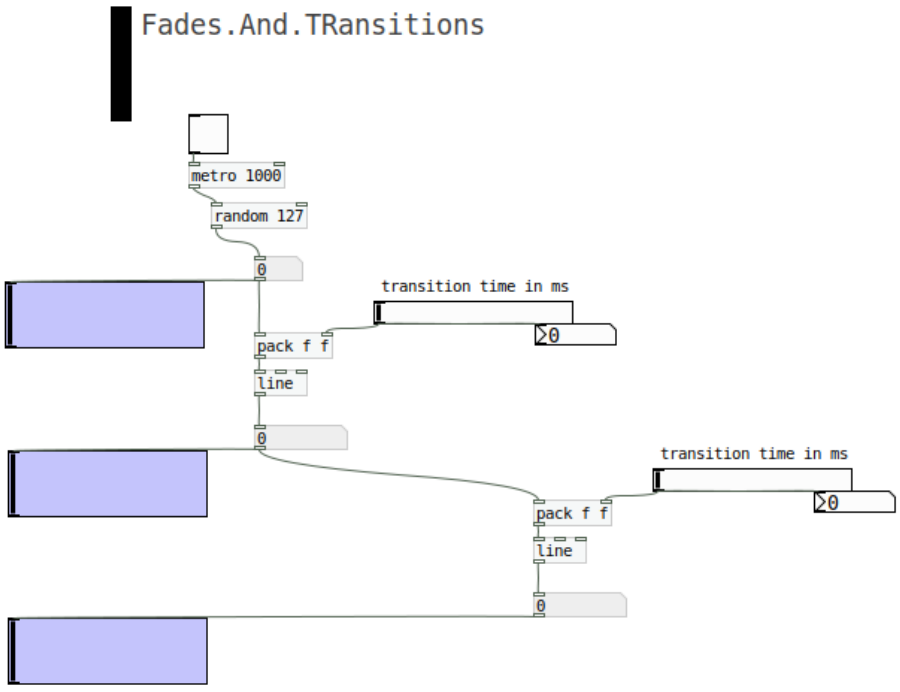




Line : Ramps and

Line is the object which makes transitions from two points.
It is useful to make several actions in the on going sound like fade ins/outs, portamentos, smooth binaural effects, among another sliding effects.

In this example a group of lines are making different transitions recoursevely from an initial changing parameter (random of 128 values every 1000ms). The 'monitoring' lilac·blue sliders are conceptually doing the same but with different sliding effect in time. Therefore the slider at the bottom behaves much more 'organic' or physical-modelled. Those sliding values we can apply to other programming parameters, like amplitudes, amounts of filters, etc.

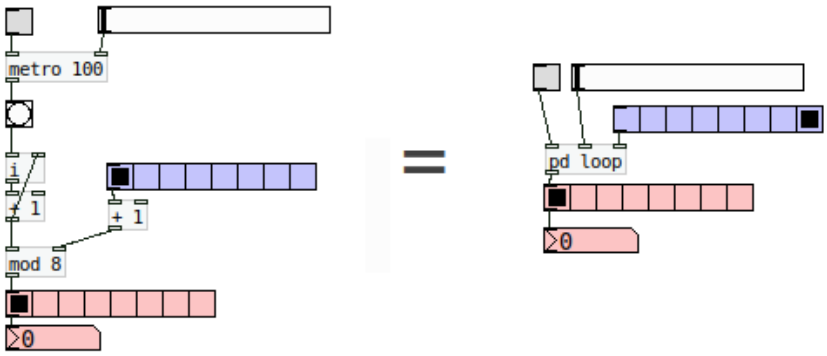


Basic algorythms
2.2. Pd Packaging Algoryhtms

One thing that usually happens is that we are programming with a lot of pieces of code that easyly fills up the GUI. One useful method to save space and code elements we are not playing as performers, is to hide pieces of code in structures that only sends/receives the dynamic parameters we want to change.
For do that, we have several options:

Sub Patch
In this example we can see two pieces of code that are exactly the same.
Is it easy to see that the right solution saves a little bit of GUI space.

packaging.algoryhtms.with.subpatches

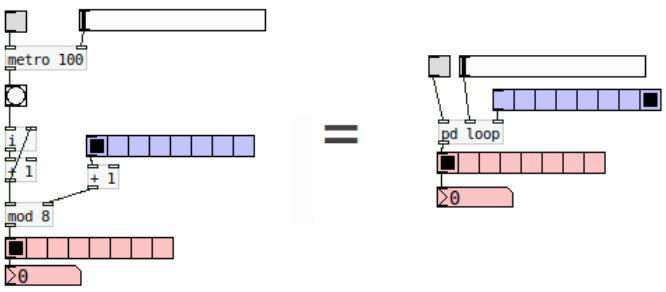


We can make a sub·Patch with the Object [pd whatever]

Is it a very useful feature whenever the original algorythm
is big in the GUI

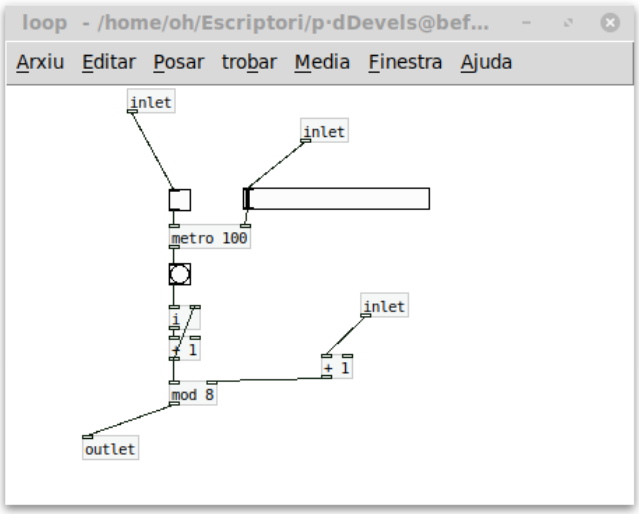
In this case we are using the subpatching method, with and object writting into : 'pd' and the name you like, in the example **[pd loop].** This creates a blank sub-patch, where we can paste the desired code and connected to the main GUI through inlets and outlets.
The order of *inlets* and *outlets* that appears in **[pd loop] at** the main GUI, corresponds to the different *inlets* and *outlets* we have written **from left to right** inside the subpatch.

packaging.algoryhtms.with.subpatches



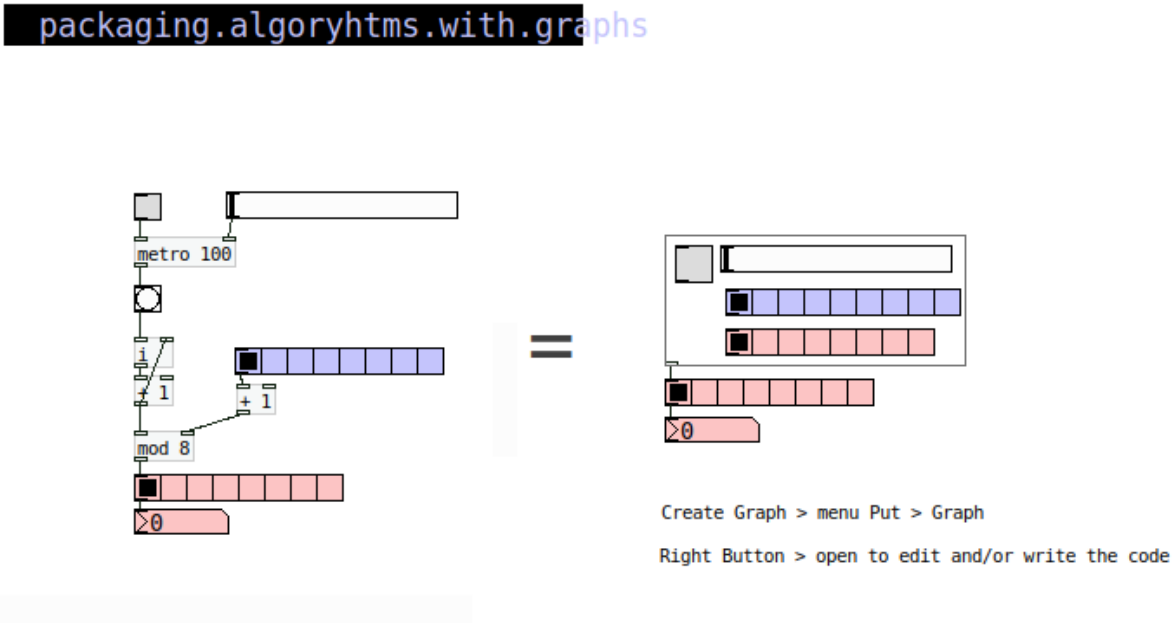
We can make a sub·Patch with the Object [pd whatever]

Is it a very useful feature whenever the original algorythm
is big in the GUI

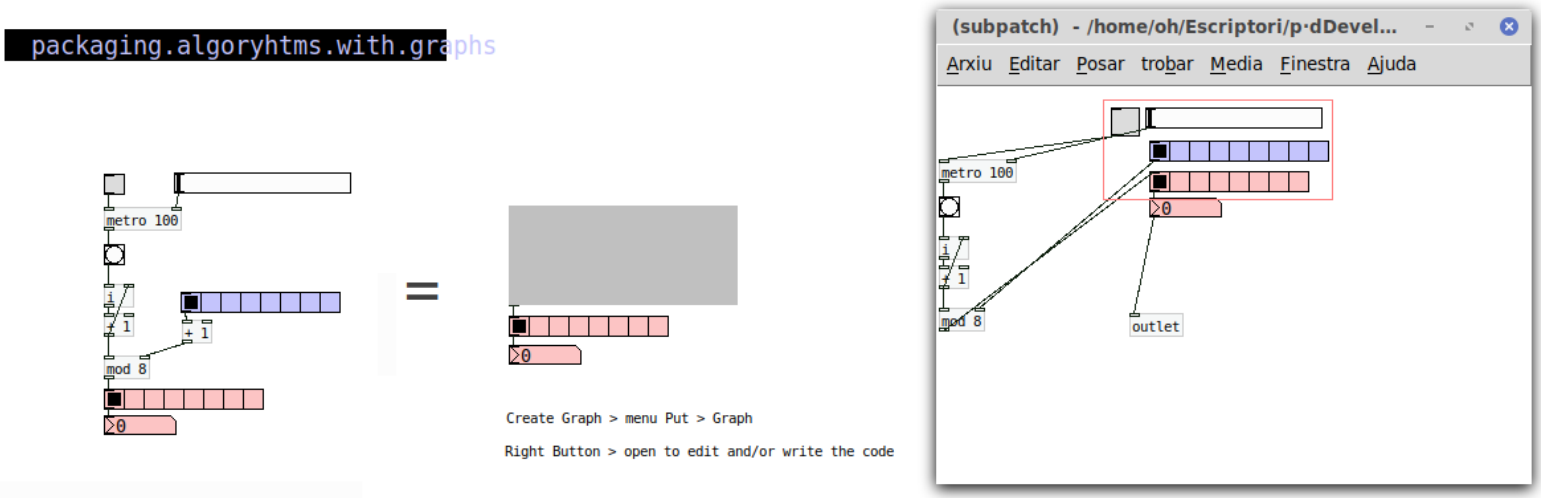


Graph

Another method to pack sections of code is with 'Graph' function > *menu put* > *graph*
As the previous method, it creates a new patch where we can 'make like a kind of hole' in it, allowing us to see the elements we want to manage.



Similarly as the previous technique, the order of *inlets* and *outlets* appearing in the main GUI's Graph corresponds to the different *inlets* and *outlets* we have done **from left to right** inside the Graph



Both methods (Subpatch & Graph) are similar, but maybe the second one helps to visually structure a main GUI for big Patches.
Anyways the SubPatch technique allows us to make complexe groups of sections of code, that we can keep in the 'backend' calling the functions or parameters we want to control, via ***sends*** *and* ***receives****.*
Sends ***[s whateverdata] [s~ whateversignal]*** *and Receives* ***[r whateverdata] [r~ whateversignal]*** *are somekind of 'wireless' connections, does not matter how deep you are (imagine you wanna send a message from the main GUI, to a subpatch which is into another subpatch into other subpatch of the main GUI etc.)*

Basic algorythms
2.3. Pd Arrays (Buffers or Memories)

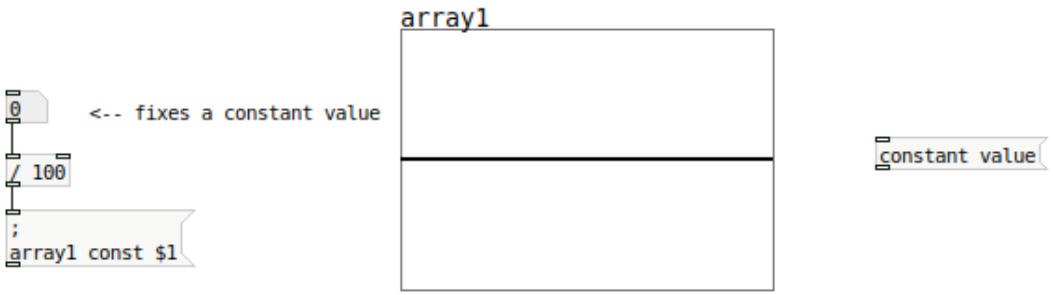
Arrays *[menú put > arrays]* are useful elements in Pd.
In fact are memory storages or buffers, where we can keep both data (numbers) and signal.
Let's start with data storages and its transformations.

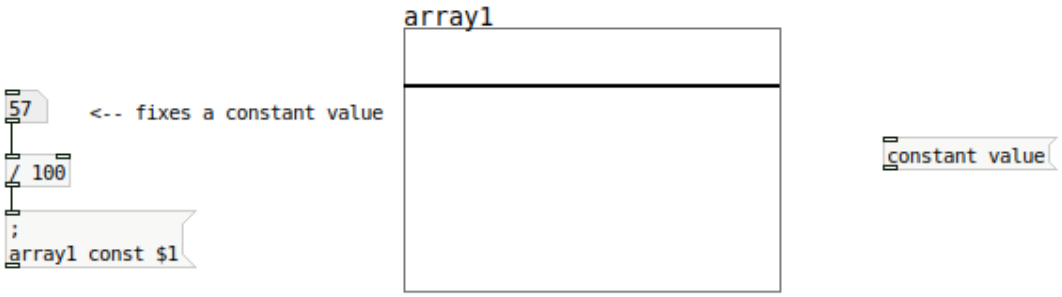
Array's transformations are usually done by specific messages, with the next conceptual syntax :

[; arrayname transformation value [

Arrays > Constant

We can use arrays to store constant values in time.
Even is not so often used, this is the syntax :
[; arrayname const value [

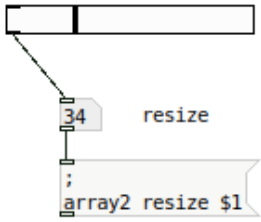


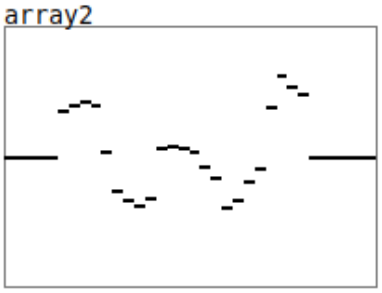


Notice that value after *const* is written with a $1.
This means that can be a **variable**, therefore a numeric value which dinamically can be changed.

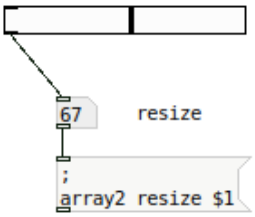
Arrays > Resize

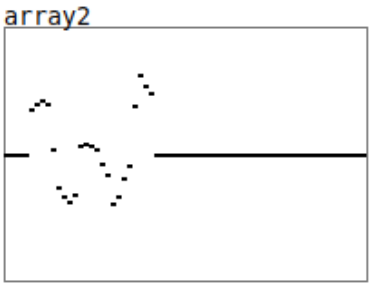
We can resize arrays in time.
This is the syntax :
[; arrayname resize value [





resize array or memory





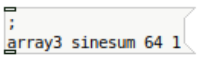
resize array or memory

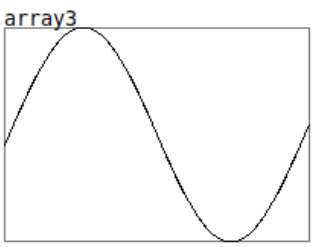
Arrays > Trigonometry functions

We can make data buffers with trigonometric structures, like sines and cosines
These are examples of the syntax :
[; arrayname sinesum arraysize string.of.values [
[; arrayname cosinesum arraysize string.of.values [

The array size has to be power of 2 > 32, 64, 128, 256, etc
The string of values you can try with different combinations between 0 and 1 > 0, 0.1, 0.3 ,0.2, 0.1, 0.4, etc..
The more large is the string, the more curvatures has the generated wave.

For instance if we want to make a basic sinewave
[; array3 sinesum 64 1 [

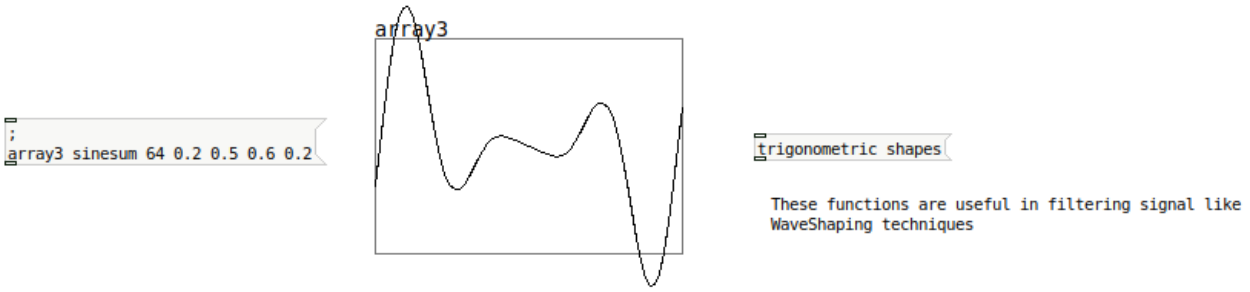


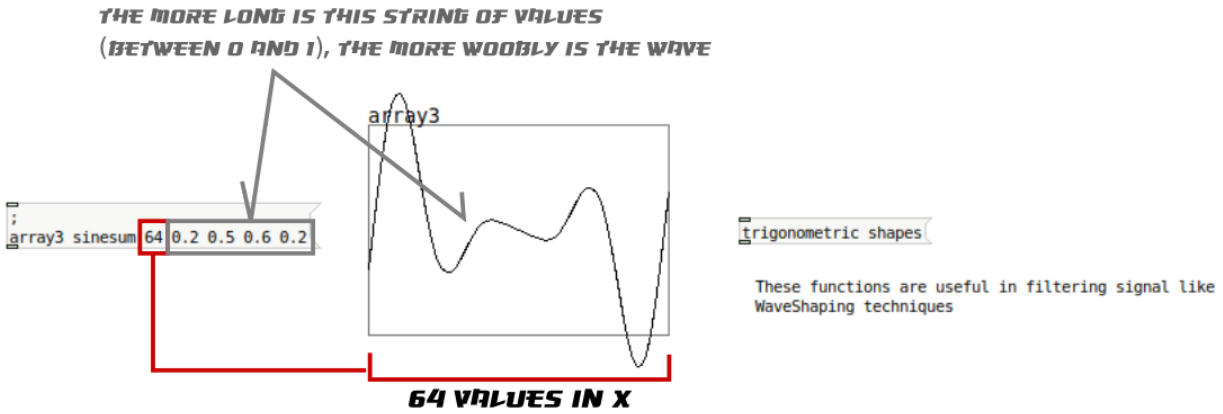


trigonometric shapes

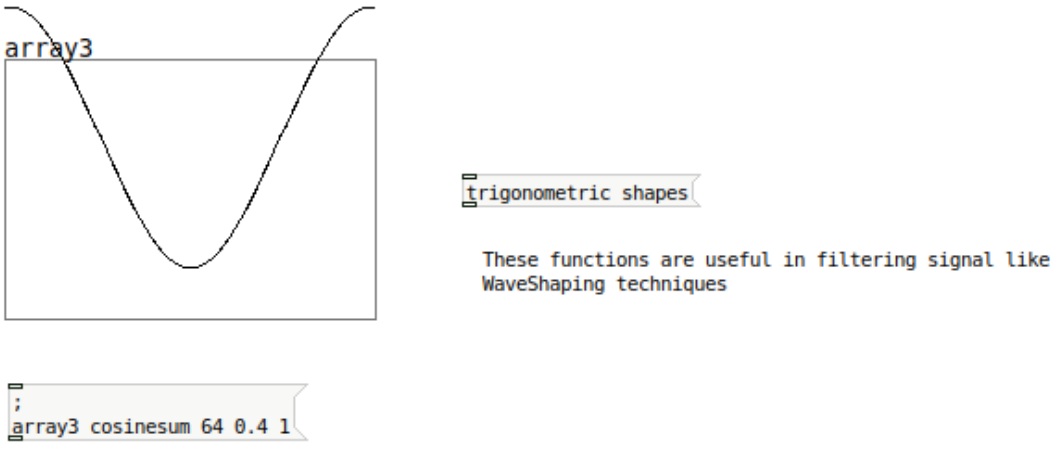
These functions are useful in filtering signal like
WaveShaping techniques

Now a sinewave a bit more '*woobly*' :
Remind that the more large is the string of values, the more curvatures has the generated wave.





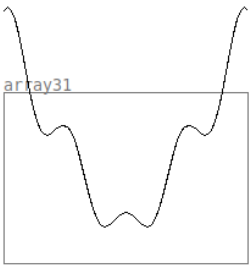
With cosines is similar
[; arrayname cosinesum arraysize string.of.values [



Arrays > Normalize

Depending on which use we want for the buffer, maybe we need to limit the ranges.
Working with sines and cosines the values on the Y axis goes from -1 til 1, but sometimes shapes can overpass these thresholds.
There is another instruction with arrays that limits this ranges, which is ‘normalize’
[; arrayname normalize value [

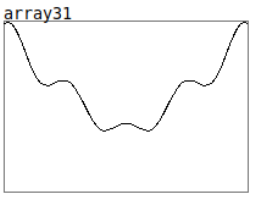
If we write *normalize 1*, shape will be fitted in -1 / 1 ranges.



trigonometric shapes

These functions are useful in filtering signal like WaveShaping techniques

```
; array31 cosinesum 64 0.4 1 0.1 0.2 0.3
```

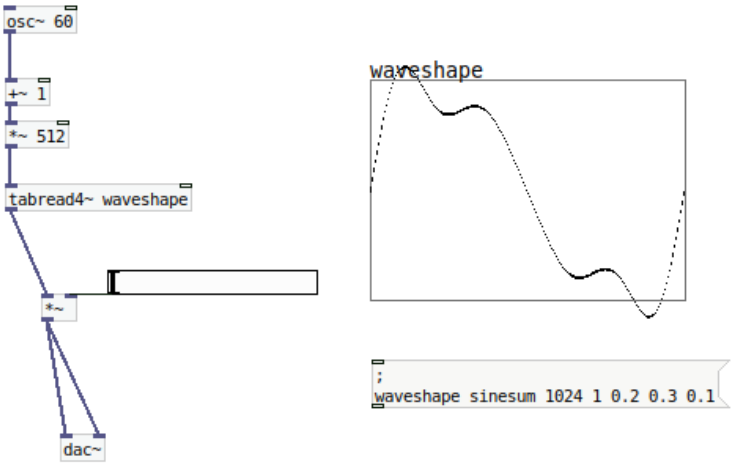


trigonometric shapes

These functions are useful in filtering signal like WaveShaping techniques

```
; array31 cosinesum 64 0.4 1 0.1 0.2 0.3
; array31 normalize 0.99
```

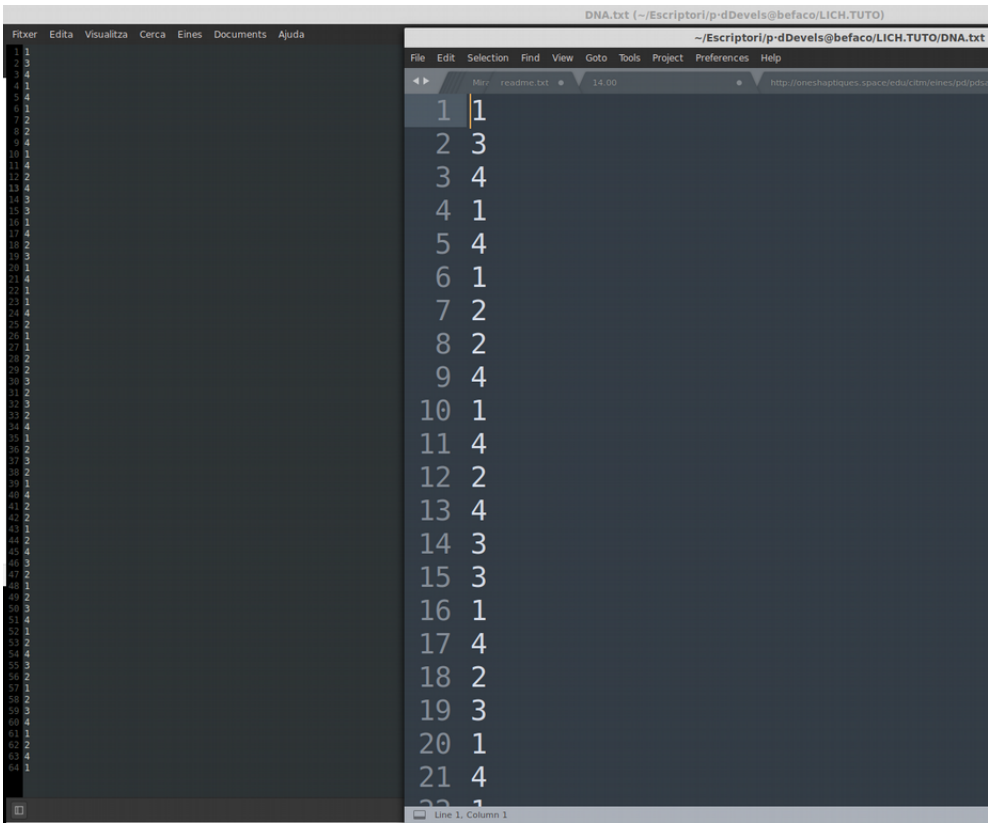
Trigonometric shapes are very useful if we want to make **WaveShaper Filters**, for example this one.



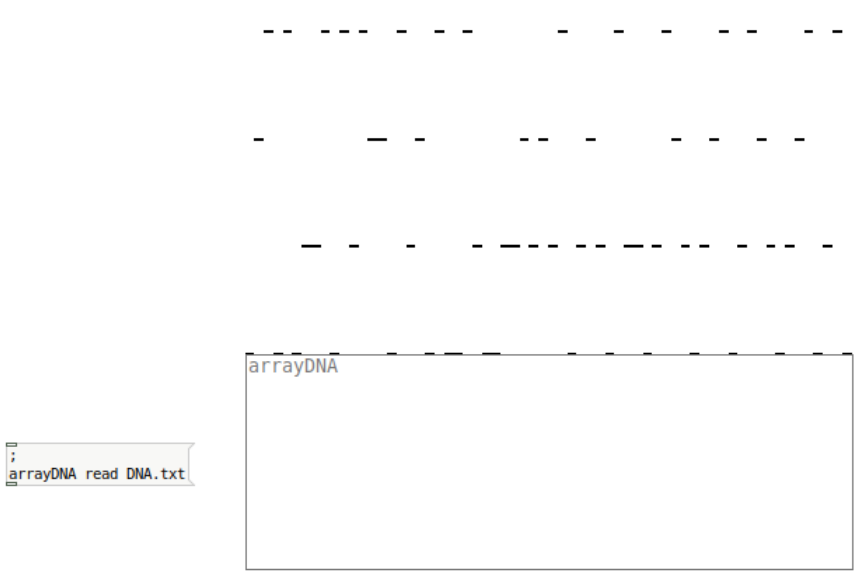
Anyways, due to the fact that it’s a nice and expressive technique of signal filtering, we’ll see different methods and tricks of it later.

Arrays > .txt
Reading and Writing TXT files

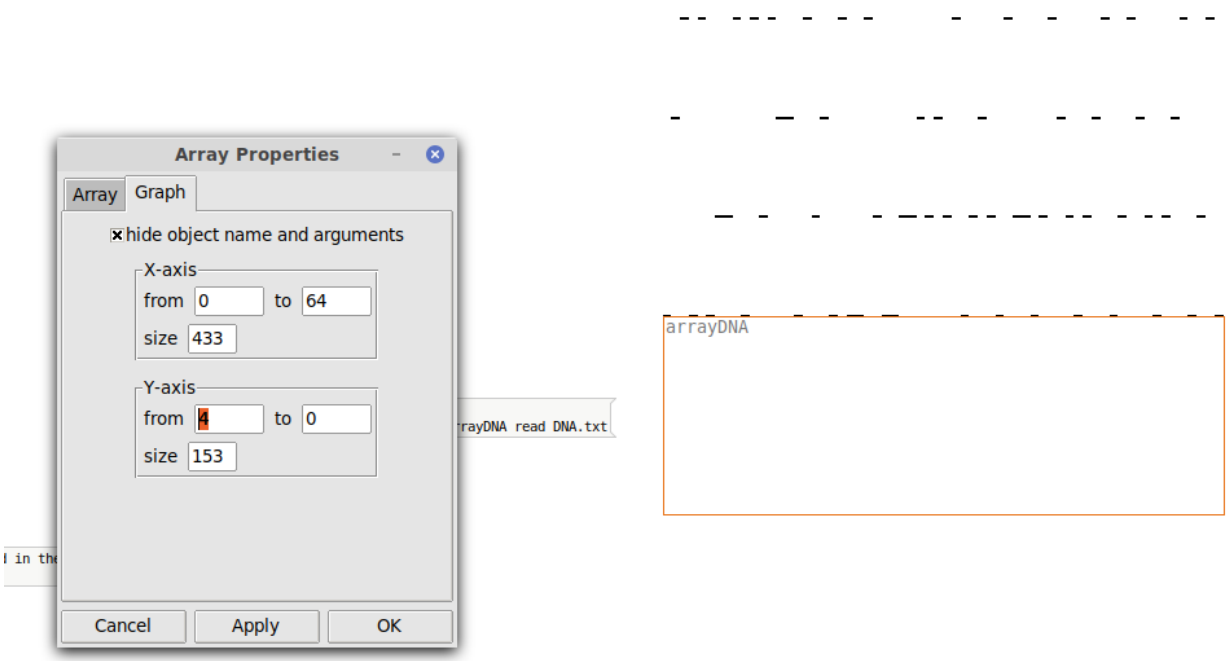
A very interesting feature with Arrays is to 'import' or grab the contents of a text file.
In fact this is an interesting feature for **sonification projects.**
In this sense, in order to Data be imported correctly by Arrays, text files has to be formated with a value per line of code.
Values has to be ALWAYS numbers, not alphanumeric structures or others, just one number for each line.
For example, this .txt file with different values between 1 and 4.
note : the image features a dual zoom of the same file.



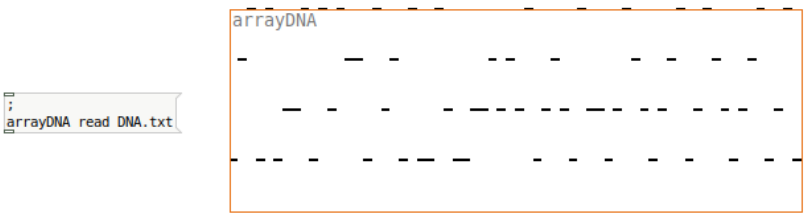
If we save this .txt file in the same directory level as the patch we are working in,
with this message,
[; arrayname read ourfilein.txt [
and bang it on it or just clicking on it :
The contents of the file will be directly transported to the array :



In the previous image, noticed that values of the array has overloaded the frame of the array. That's because, by default Arrays generates a score with values from -1 til 1.
Due to the fact we have values in the .txt (and therefore in the array) from 1 to 4 it features an overloaded representation.
How can we rearrange it?
If we make right button over the array and click properties, we'll see a pop up menú where we can change different parameters : size of the Array in samples, Size of the Array in pixels (X & Y), range of values in Y axis, size of the Array itself amontg other features of visualization.

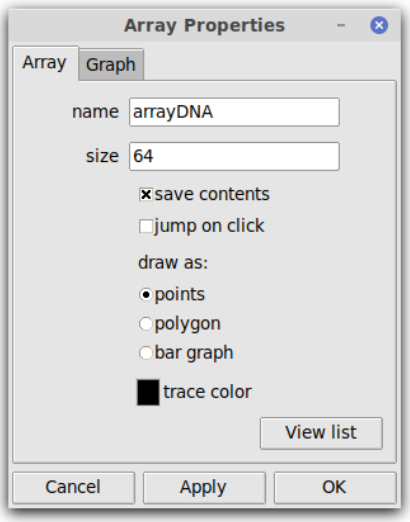


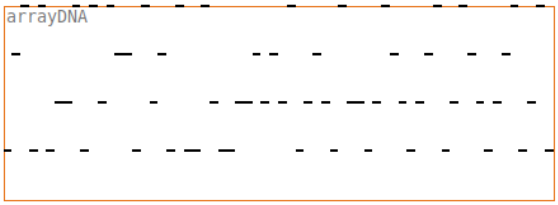
Once set up values from 0 to 4, Array is rendered like this :



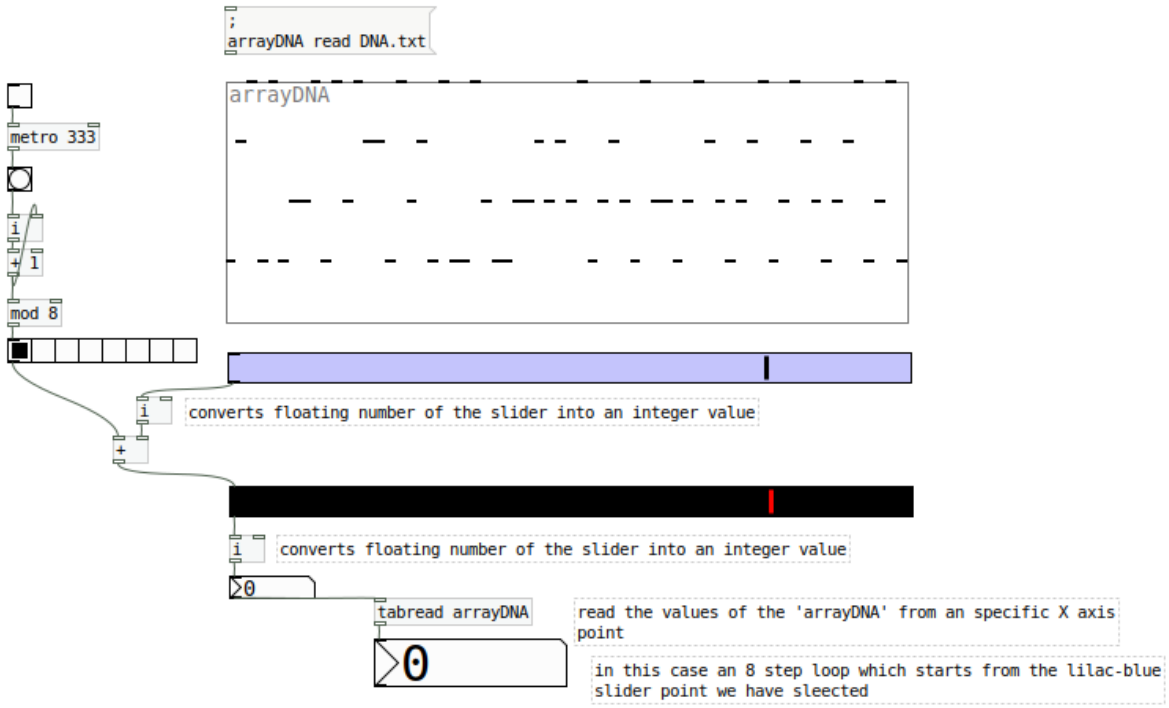
Also another trick and very interesting Array's feature is to **save the content.**
Imagine in an array we have a sequenced bassline midi notes. Every time we open the patch, the contents will be erased and we have to bang it to a certain message to loading it again, unless we have already set up **'save contents'** in Array Properies **.**
If we do that, every time we open the patch the content of this Array or memory will be kept like a list of values embed in the array and therefore in the patch (file.pd) we are working in.
This method works both for *data* storages and *signal* storages within the arrays.

>>>>>>>>>>>>>>>





Playing around with arrays and .txt files allows us to build **sequencers** in a pretty easy way, for example :



Anyways we'll see later in the sequencers chapter of this tutorial.

Sampling
3. Pd Sampling

After we have seen different array methods, let's remind that Arrays we can use it for storing numbers, but also for storing signal. (In fact the second one is also a data storaging but it has some specific details to take it into account, to manage it as a signal).

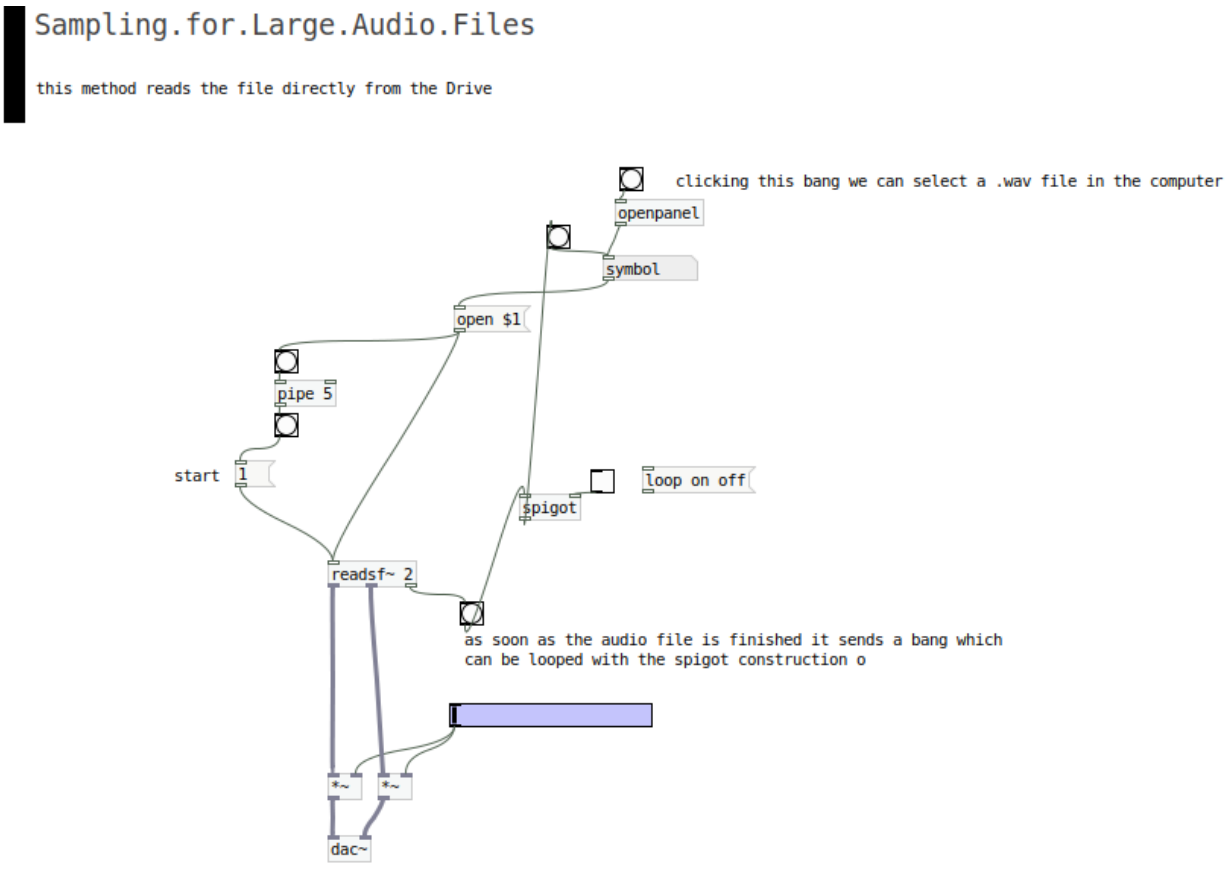
If you are a musician, you'll already know what **sampling** is, but as a short reminder, is a technique which uses audio fragments to play and process them.
Notice that in Pd and in DSP techs in general, sample is not only the concept of storaging audio signal in buffers, but also the 'pixels' of information which describes a particular storaged signal. Therefore for an specific unit of time, the more samples of information we have, the more detailed will be the signal. That's interesting for reproduce preexisting audio files, but for managing signal calculations in real time, sometimes the HD quality render effort is collapsing with the processor speed, and therefore its more efficient to balance and tweak it, in order to get a nicer DSP performance.

Let's see different methods to play an audio file from the computer.

[readsf~]

The first and most simple method is to read the audio file (or in .wav or in .aiff) directly from the disk (therefore without buffer it). This method is required for large* audio files we want to play.

*large meaning audio from 30 seconds until hours.



[readsf~] as a memotechnique : **reads**ound**f**ile
is an object where we can describe the number of channels we want to play :
for a mono audio file **[readsf~]**
for an stereo file **[readsf~ 2]**
...and so forth until 64 audio separated channels.
But also we can add another argument in the object to define some more specific details like buffer size in bytes per channel for specific purposes [not often used].

Therefore the syntax :
[readsf~ numchannels buffer.size.per.channel.in.bytes]

In the object [readsf~] will be several outlets. The firsts from the left corresponds to the channels that has been described in the object. There is also another outlet on the right bottom which trigs a bang as soon file has ended the whole reproduction. Therefore if audio sample is correctly edited we can make easyly a looper.

In the previous example due to the fact we have been using a symbol (menú put > symbol), the first time we loaded the file from the HardDrive the path has been storaged in the symbol. So anytime we want to make a loop the sequence of triggers first trigs the path and after 5 ms [pipe 5] trigs the [1] message which starts the play.*
The object spigot is in charge if we want to make this flux like a loop. So spigot is in fact like a gate that is opening / not opening >> flux continuity / not flux continuity

**pipe is a delay of data very useful to scheduling instructions and dataflow.*

Recording : [writesf~]

The opposite function of **[readsf~]** is **[writesf~]** that is an object which can capture or record any signal that is throwing into it.

[osc~ 111]
|
[writesf~]

That's an interesting feature if you want to **record** a performance of your patch within the DSP.

Remind that in order to record some running audio is important to follow an specific order :

1.**write the name** of the file we want to create in a message associated to writesf~
[open /your/path/Desktop/record.wav [

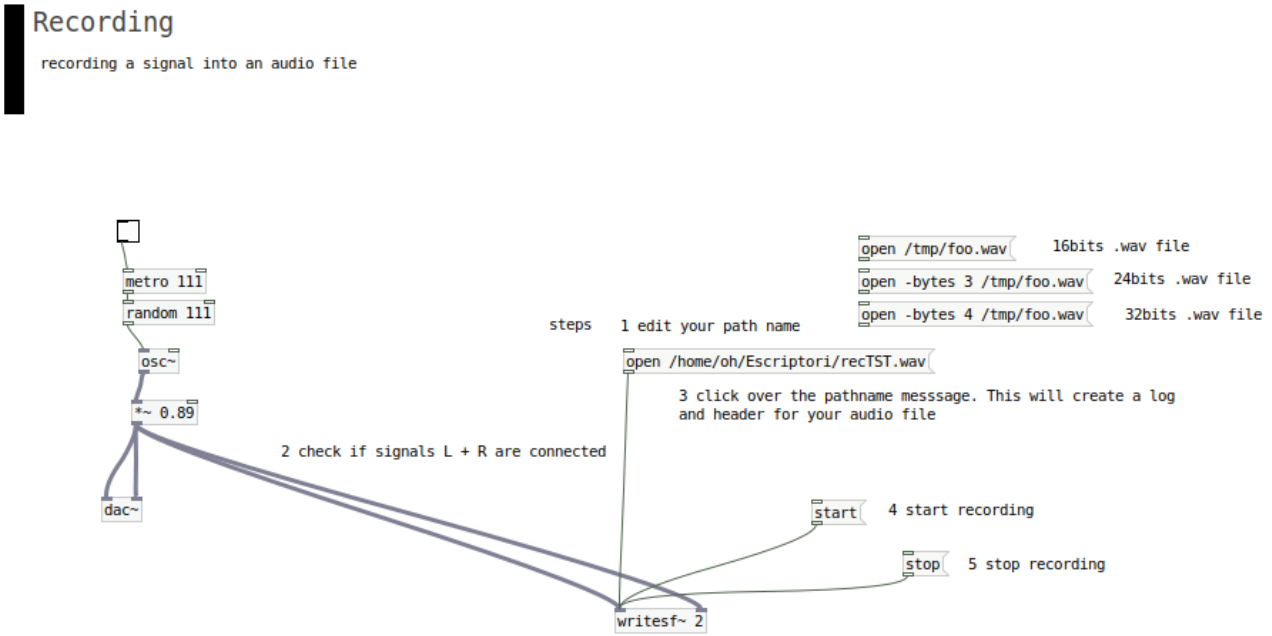
*2. **connect whatever signal** you want to record into the writesf's channels you wanna record (L+R if is [writesf~ 2]).*

*3.If you are ready to record, then **Click** on the message created before **[open /your/path/Desktop/record.wav [***

*4.click on **[start [** message, to init the recording*

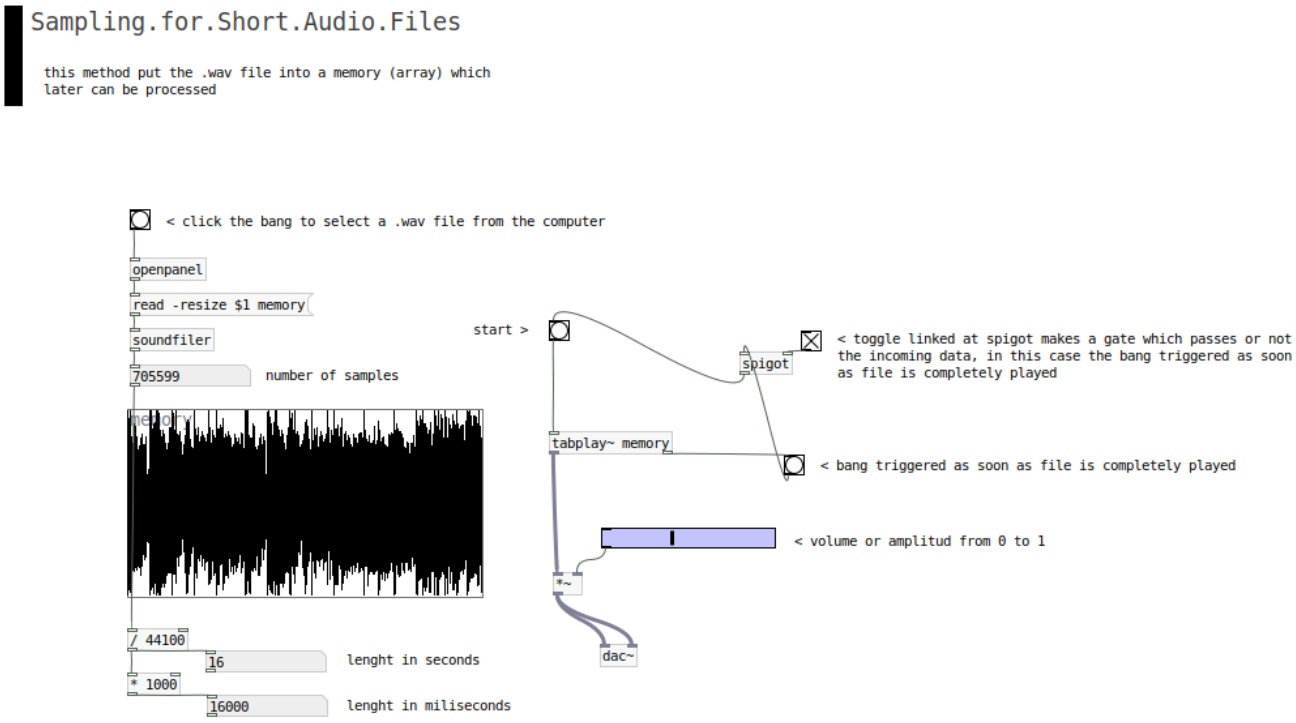
*5.click on **[stop [** message, to stop the recording*

*note : clicking on the message with your path and name.wav it creates a **log** file that iniciates the recording with the instruction **[start [**. Therefore if we do not click on the message, log will not be created and Neither the audio record.*

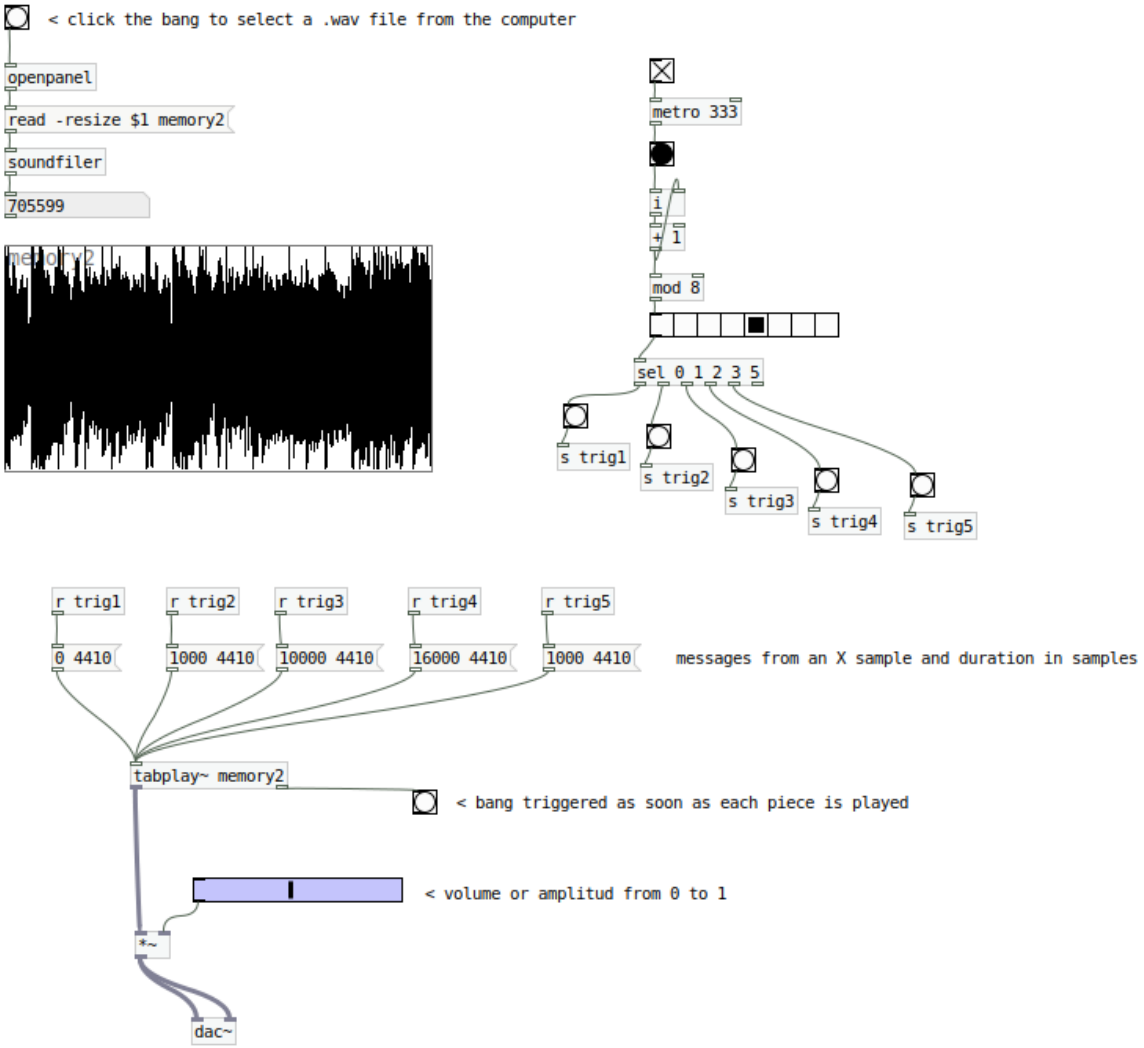


[tabplay~]

maybe the most precise method to sampling in the classic sense is with the object [tabplay~]
First we have to load a .wav or .aiff file into an array or memory that after we can trig, and even loop it.
Notice that in order to adapt an audio file into an array it is necessary the object **[soundfiler]** and it previous message **[read -resize $1 arrayname [**



With **[tabplay~]** we can play the whole sample like in the previous example, but also we can make another use of it that is *slicing*.
Slicing is a reproduction of a particular section of the original file that we can manage with messages that defines the init point of the sample and the lenght in samples of the slice > **[0 4410 [[100 4410 [** etc.
In the next figure a tiny sequencer of 8 steps triggers several slices.



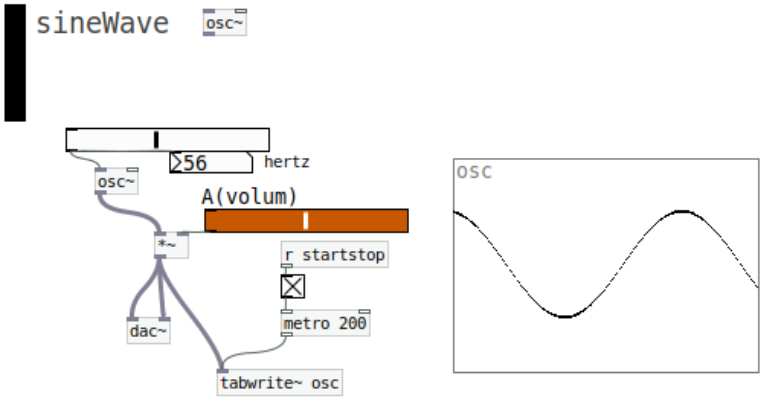
Synths
4.1. Pd Synths Waveforms

If you are an electronic musician you can jump this introduction :
Synthesizers both in software and hardware are beatiful instruments to reproduce a massive range and types of sounds, from the imitation of natural sounds until the production of unique and particular sounds.

All this massive range of sounds, can be produced by a vast number of techniques and methods, but anyhow we can build any type of synthesizer from particular ‘sonic bricks’, that afterwards we can combine, organize and filter in complex structures.
These bricks are refered as Waveforms driven by Oscillators.
An Oscillator is one of the most basical conceptual element in a synthesizer :
In hardware framework, it translates electricity to an oscillating acoustical signal.
In software framework, are functions which translates data into an oscillating acoustical signal through the DSP.

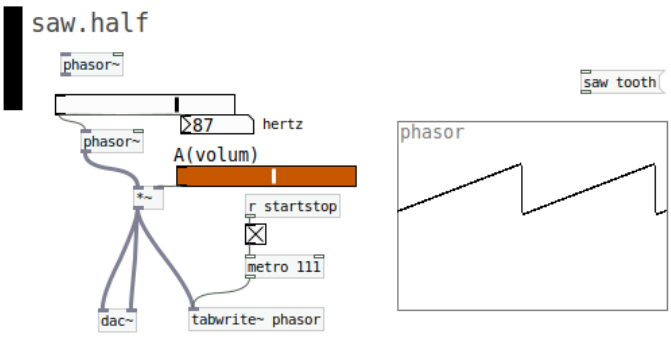
In pd there are some native waveforms and some others that we can build.

Sinewave Oscillator > [osc~]

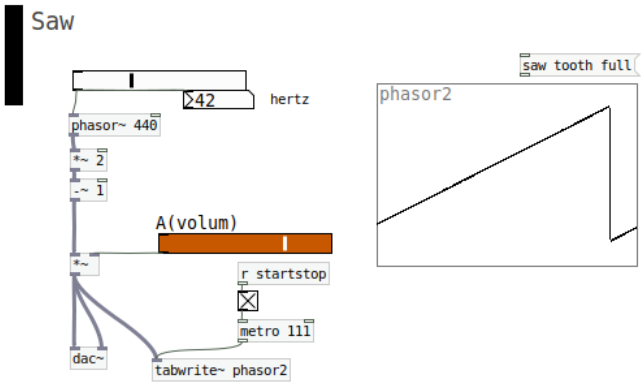


Saw Oscillator > [phasor~]

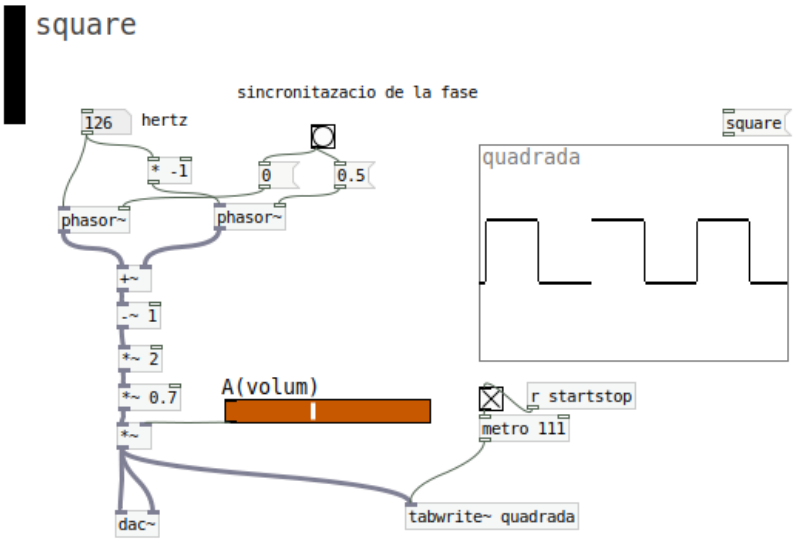
(half) saw >



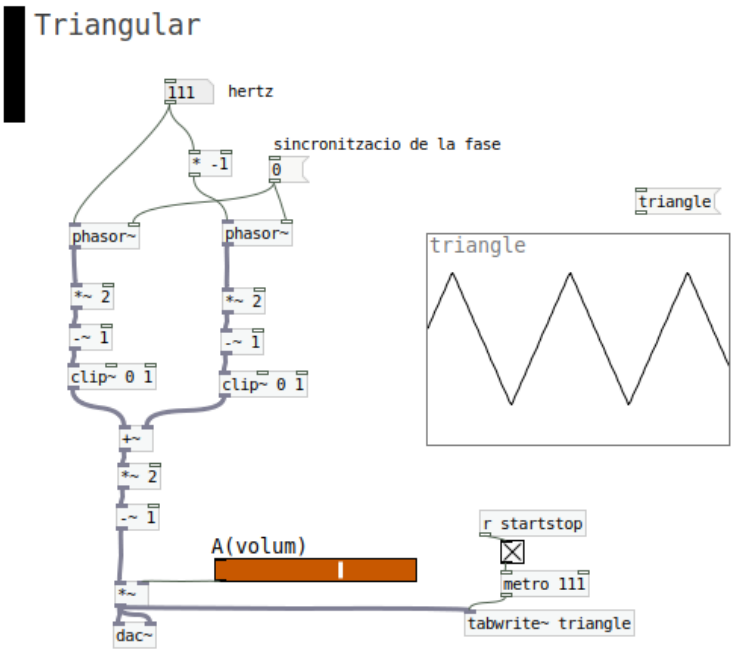
(full) saw >



Square Waveform

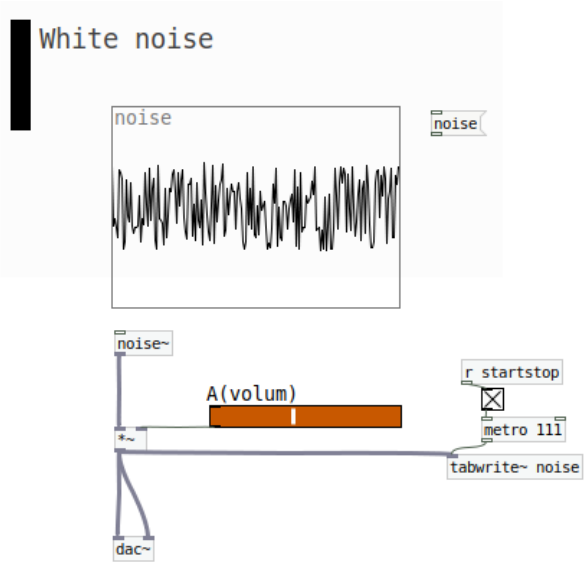


Triangular Waveform



Noise Waveform

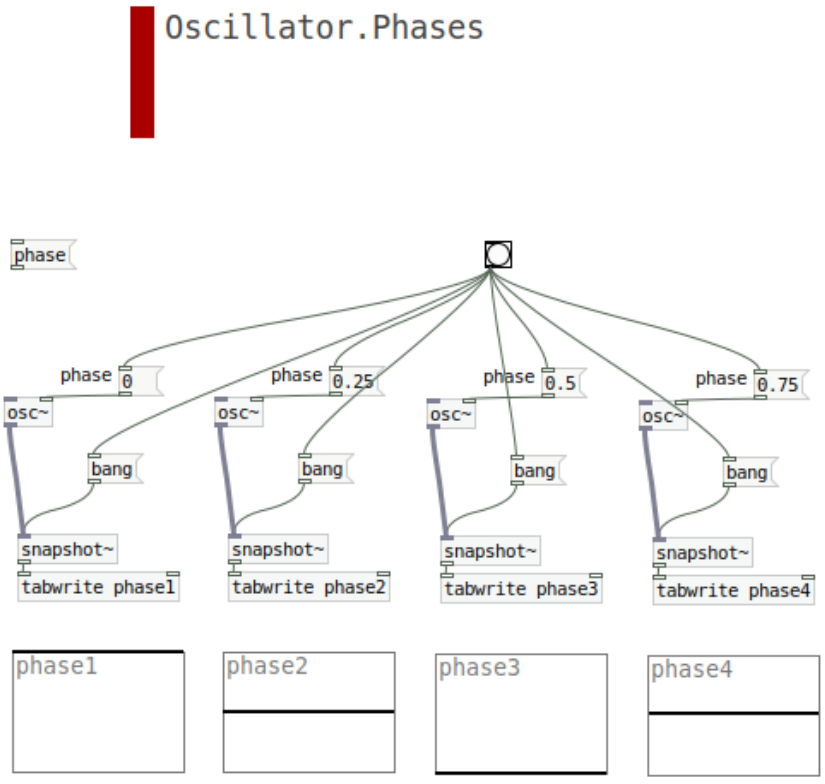
Even is not exactly a waveform type, white noise **[noise~]** it is a very common *'brick'* in Synth's World. White noise is a pretty particular sonic element in which all frequencies are reproducing at the same time. For this reason is a very used element in substractive and percussive synth techniques.



Synths

4.2. Pd Phases in Oscillators and Phasors

Phases can be set both in **[osc~]** and **[phasor~]** objects through the right inlet.
Notice that degrees are represented in a range from 0 to 1

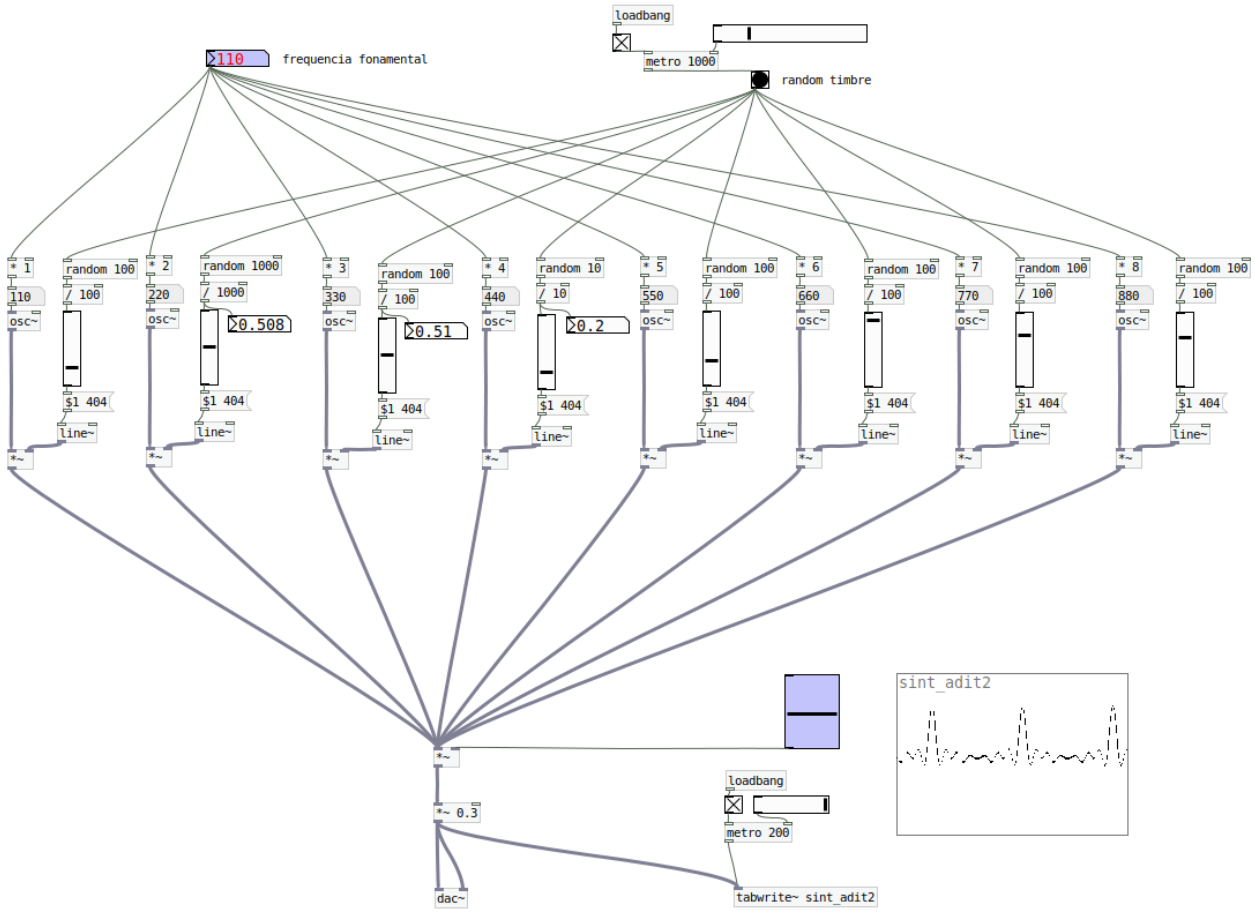


As you may know there are a big amount of synthesis types.
However we differenciate among different classical categories like additive, substractive or granular.
Another categories and techniques of synthesis can be based in *physical models*, in *probabilistic models* (stokastic synthesis), or imitation of sounds like *formant synths* imitating human voice, among others.

Synths
4.3. Pd Additive Synthesis

Visual Domain Analogy > Drawing different colors and shapes in a blank and white canvas.
The additive synth would be the whole picture.

Additive Synthesis : several layers of generated sound combining and interacting between them.
In the next example, a group of 8 oscillators with superior harmonics (multipliers) is changing randomly the amplitud of each line before mixing them. The result is a continuous tone / drone in which timber is changing all the time.

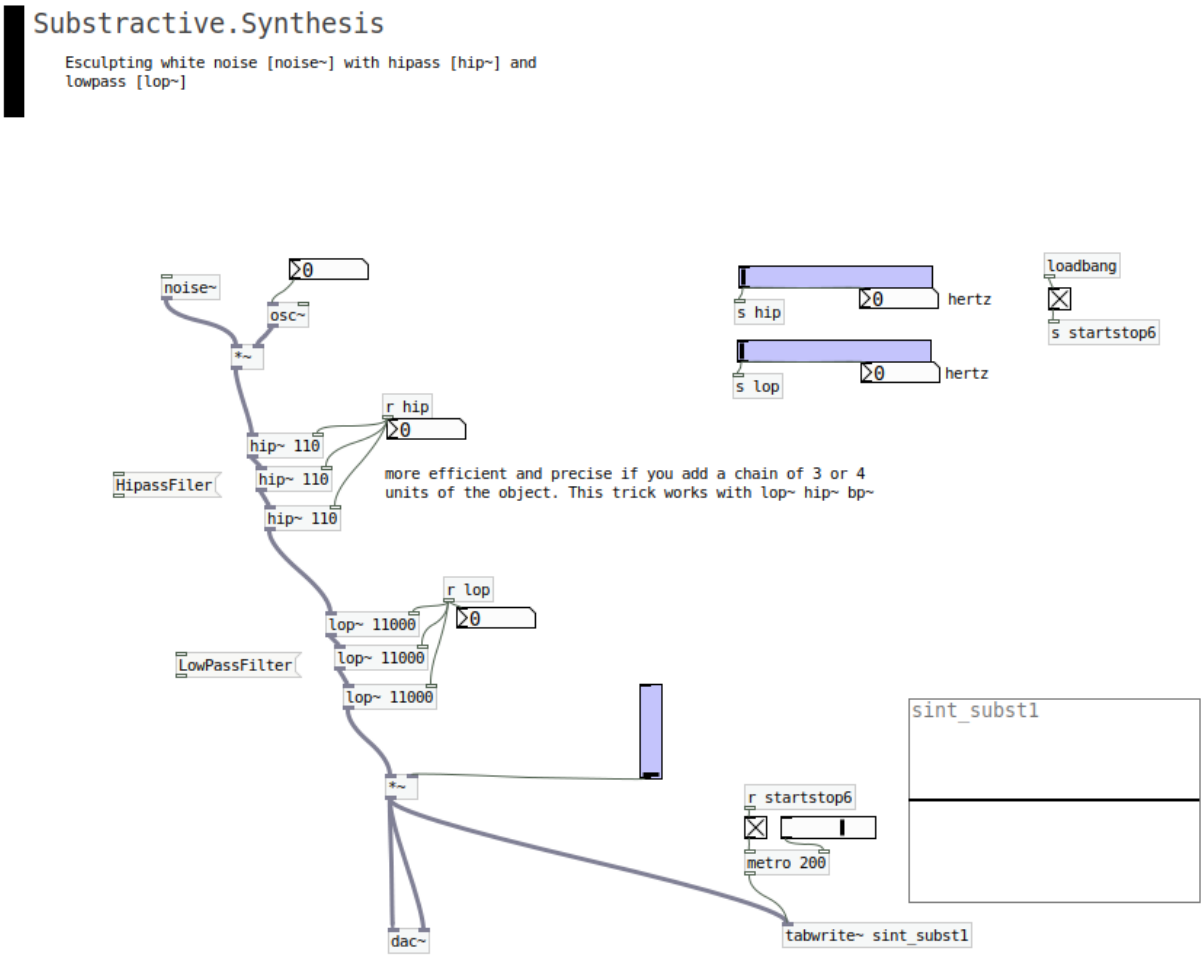


Synths
4.4. Pd Substractive Synthesis

Visual Domain Analogy > carving with different shapes and sizes the surface of a black canvas, sculpting it and arriving to the canvas basement. The Substractive synth would be the whole sculpted picture.

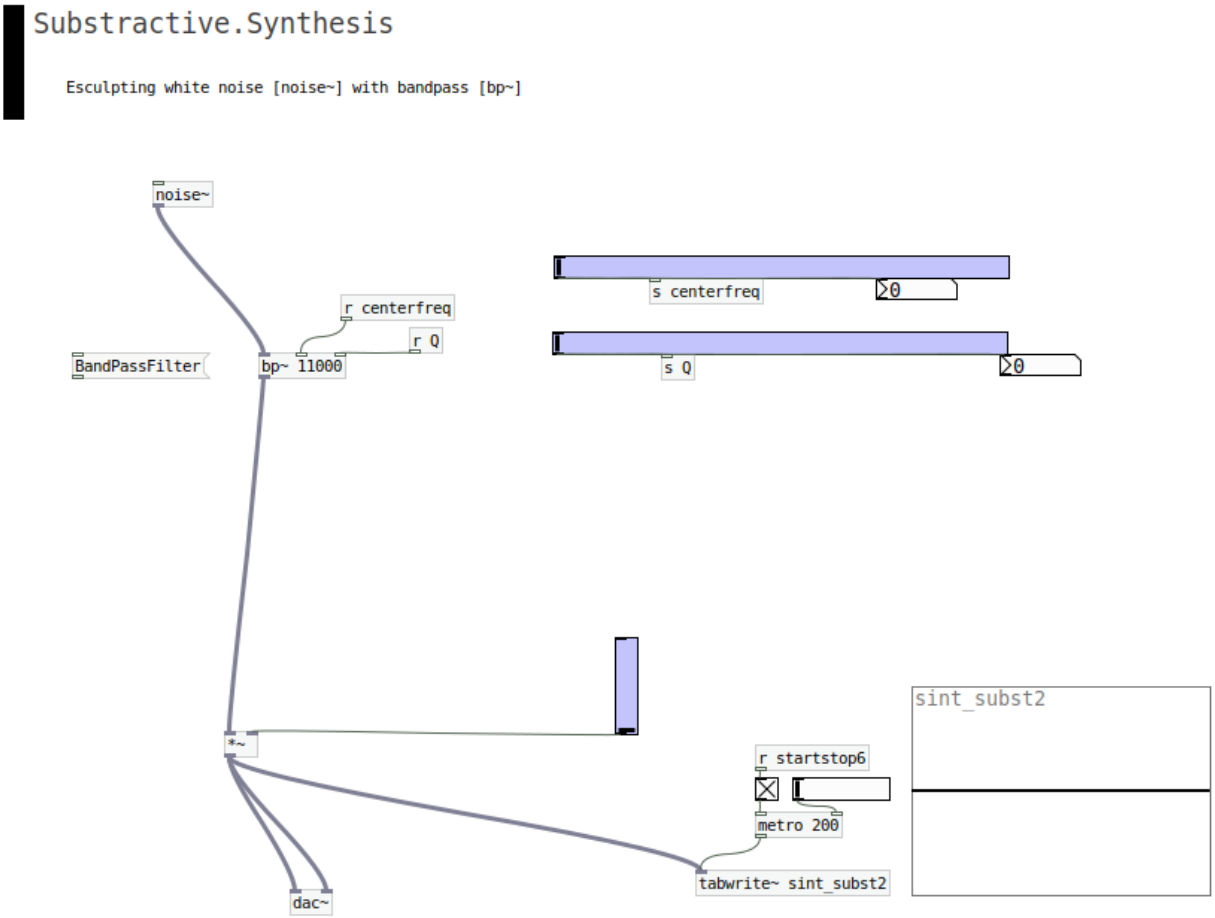
Substractive Synthesis : sculpting with different kind of filters a massive block of sound generated usually with white noise [noise~].

In the next example an oscillator is modulating a white noise which afterwards can be filtered with hipass filters or lowpass filters.
[lop~ value.in.hertz]
[hip~ value.in.hertz]



As a reminder for newbies:
*A hi-pass, features the whole auditive spectrum **starting from** the hipass parameter or threshold.*
*A lo-pass, features the whole auditive spectrum **until** the lopass parameter or threshold.*

In the next example a white noise is filtered with a band pass filter.
[bp~ value.in.hertz]



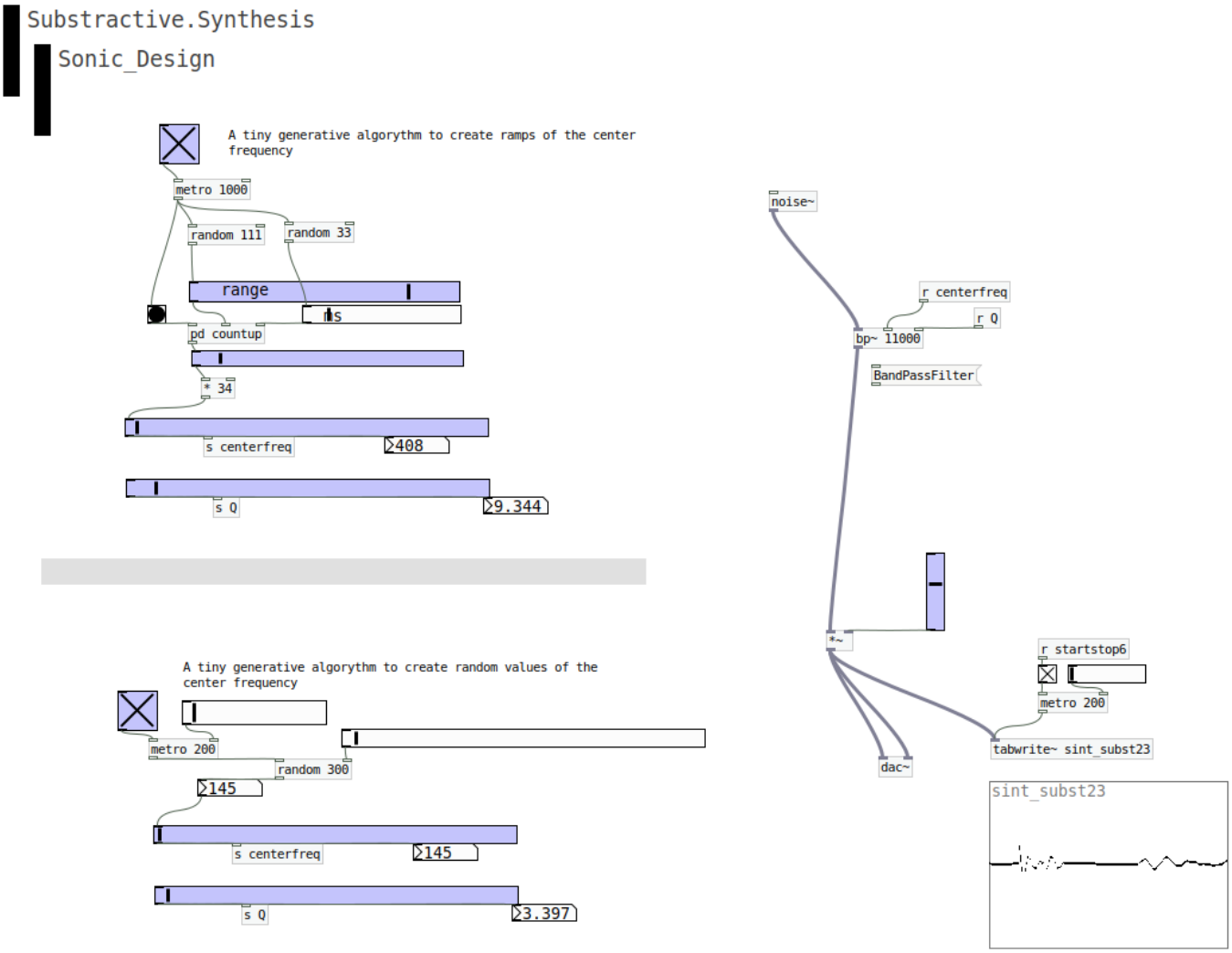
As a reminder for newbies:
A band-pass, executes a 'mountain-shape' filter from an incoming signal in which for a certain frequency defined in the bp.
The 'mountain shape' can be more or less vertically stretched depending on the amount of Q parameter, producing more or less ressonant effect, according the reflections inside the cavity of the 'mountain-shape'
Therefore:
High values of Q represents a kind of huge and vertically stretched mountain, and will be MORE ressonance on the frequency range defined by bp~.
Low values of Q represents a tiny hill wider in the bottom, and will be LESS ressonance on the frequency range defined by bp~.

//////////////////////

The previous examples may be a bit obvious and ‘nothing’ special as a synths, but may be can constitute the basis for other operations and tricks in time.

For example, with the band pass example, we can introduce a couple of tiny algorythms to create some particular effects.
In the first one (top left), some kind of sliding or *portamento* effect is produced in the band pass frequency. Every second is doing this effect with a random frequency in a different amount of time, therefore a slightly different action within a repetition process.

In the second one (bottom left), a random generator produces different sudden frequency values.
This sudden changes produces some glitches with a particular sound-ressonant bubbles effect.

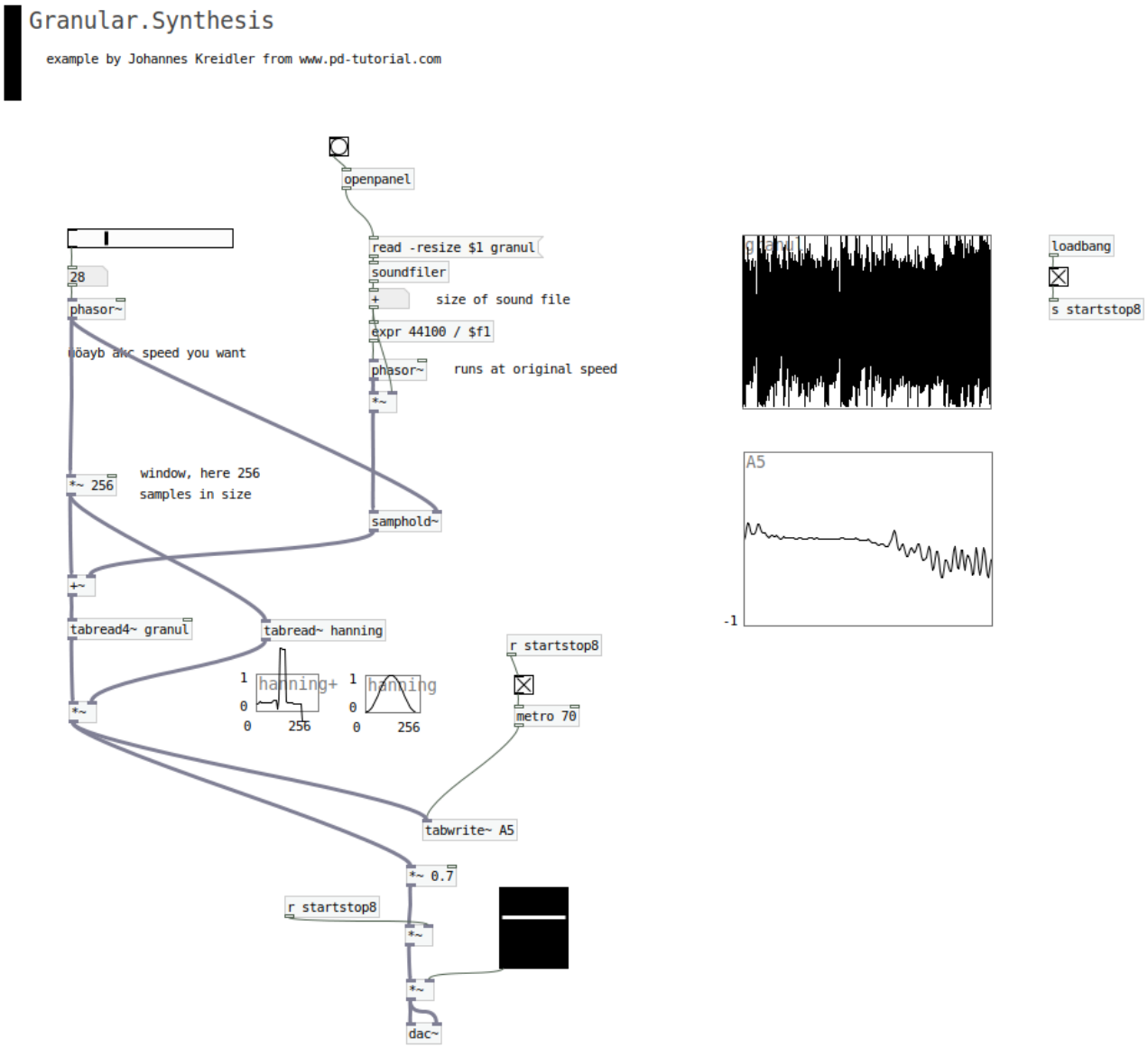


Notice that those algorythms are a calculus layer over the signal structure. Therefore it is possible to combine and reproduce several algorythms at once, producing expressive and unexpected effects.

Synths
4.4. Pd Granular Synthesis

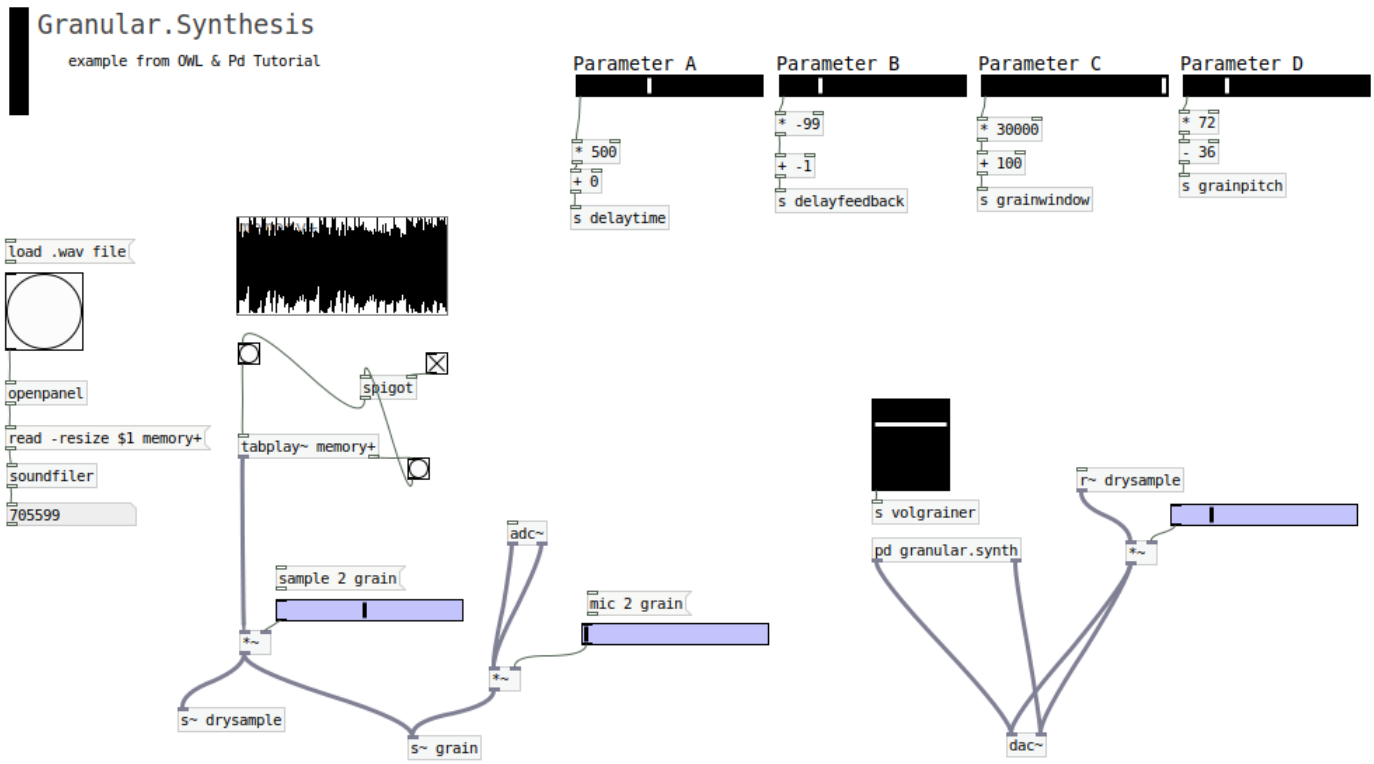
Another interesting technique, originally conceived from digital music systems, is granular synthesis.
Granular synths are somekind inspired by quantum physics applied to sound.
In this technique a particular audio sample is reprocessed like if we would have a microscope targeting over a tiny sample slice, with different non-linear parameters to tweak like amount of particles or *grains*, asyncrony of them, among other non conventional parameters.

In pd Vanilla , that is the version in which we will be able to compile for LICH module, granular synths are not the most complexes, but at least we can use it to stretch and distort samples as an expressive sonic resource.



Notice that these techniques requires an advanced level of DSP Programming skills, that often as a musicians, or coder-musicians we use them just with few tweaks.

Another example of granular synth is the next one, borrowed from OWL rebel Tech repos :



If you like this kind of synthesis and want to master it, the book [‘MicroSound' C.Roads](#) *is a very nice reccomendation.*

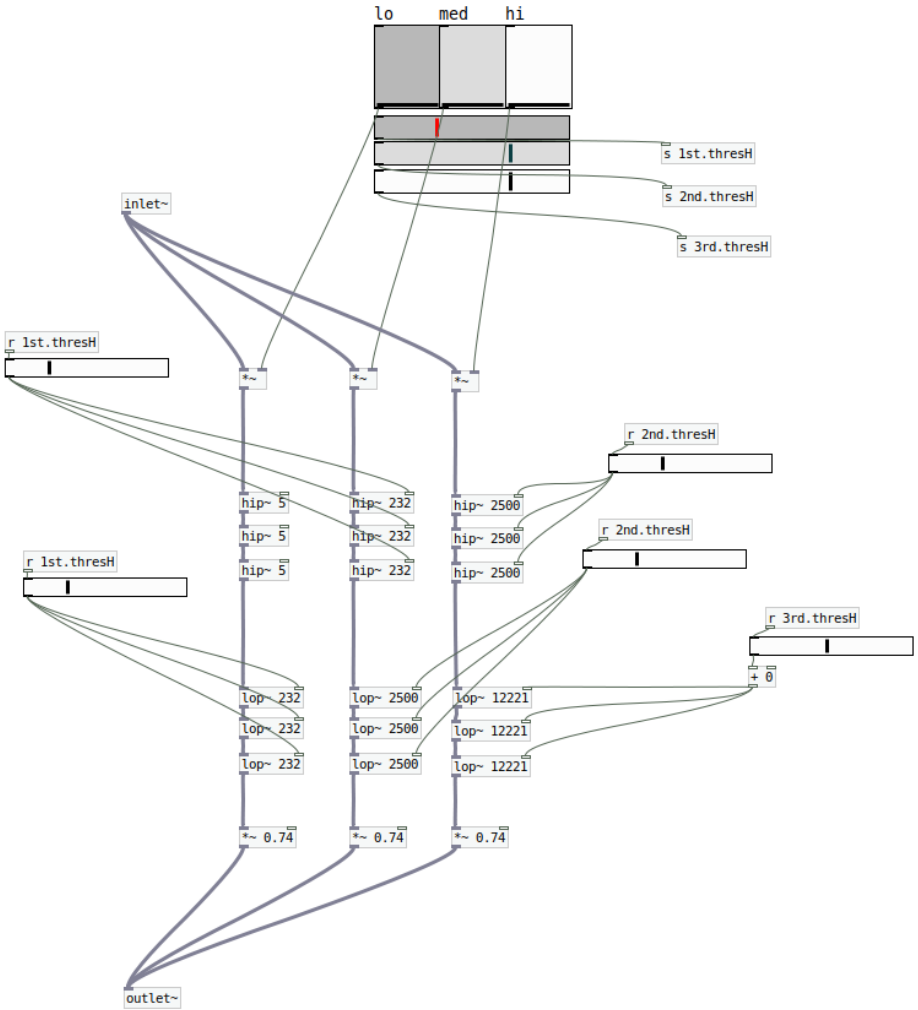
With synthesizers we can produce synthetic sounds with the most fundamental bricks like we already saw, but also is a lot of fun to model, transform and sculpt any preproduced synthetic sound. Thats the process of signal filtering.

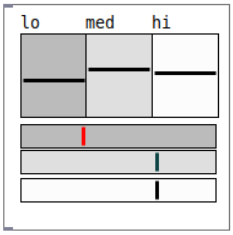
Filtering Signal
5.1 Eqs :// hip~ lop~ bp~

Like we already saw, we can filter any signal through the classic filters lopass **[lop~ freq]** hipass **[hip~ freq]** and bandpass **[bp~ freq Q]** .

If we want to build an Equalizer we can split the main signal into the EQ channels we want, and drive each line with the appropiate **[hip~]** and **[lop~]** thresholds.
In addition in the end of each line, there is an attenuator **[*~ 0.74]** due to the fact that signal line is triplicated.

3.Band.EQ



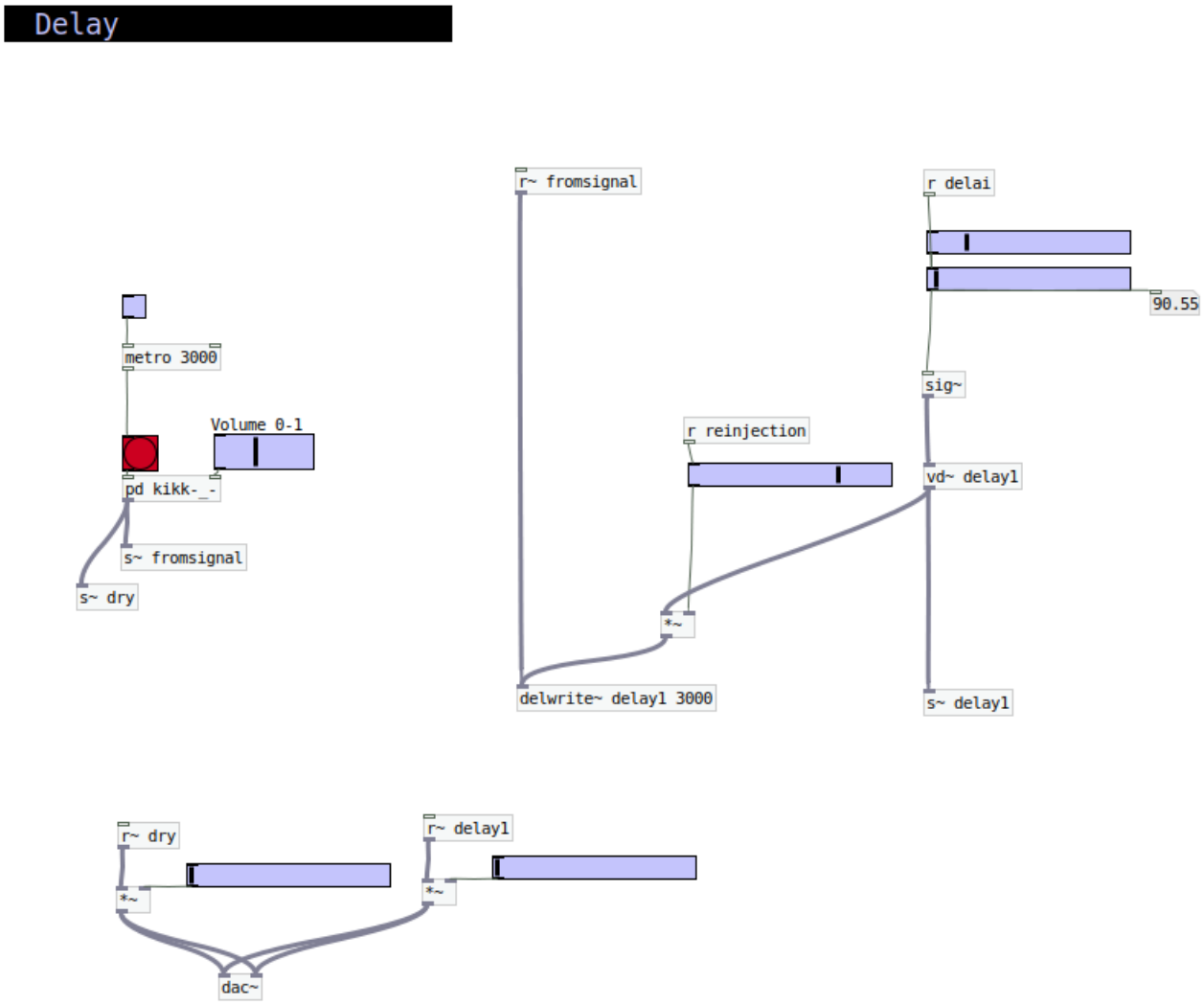


Filtering Signal
5.2 Delays

Maybe **Delays** is one of the most classic effects (but at the same time essential) in signal processing, due to the fact that is not affecting directly the character of the sound, but is affecting it in the time domain.

In order to build delays we need to send any particular signal into a buffer with
[delwrite~ name.of.delay default.time]
and later call it with **[vd~ name.of.delay]** to the main signal mix. The values inside this object has to be managed with **[sig~]** which allows to introduce numbers (data) into a signal object. Therefore is a method to dinamically introduce delay times into the **[vd~]** object.

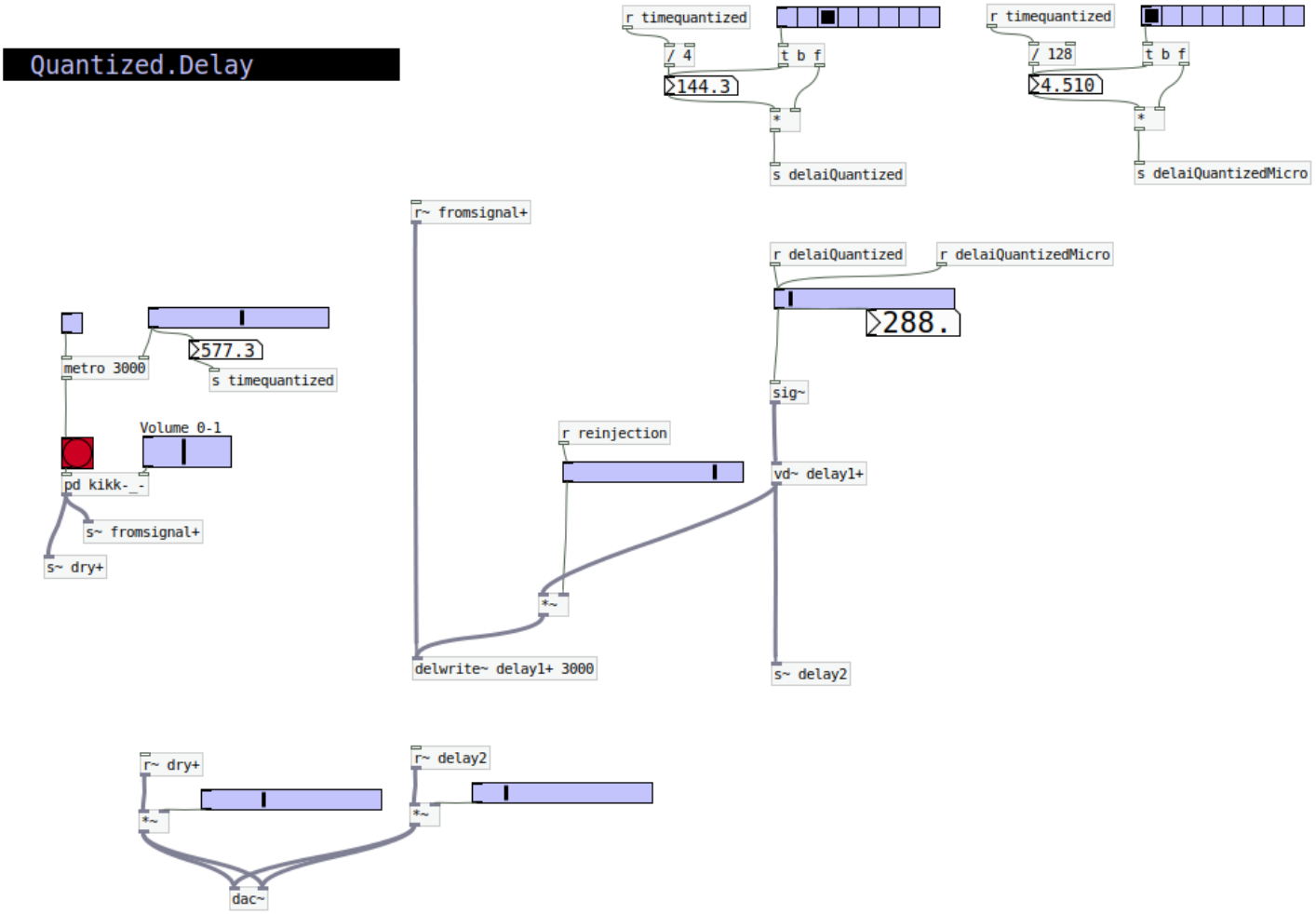
In addition we can reinject the processed signal's delay into the original loop through a gate **[*~]** allowing to reinject signal from 0 to 1 (zero reinjection to full reinjection feedback).
Note: if you use delay's feedback in LICH applications, feedback values has to be until +- 0.75 because firmware doen not support calculations fro higher feedback values.



Another interesting feature with **Delays** with Digital Processing techniques, is to **quantize** the delay times related to a certain master clock we are running.

Quantized Delays

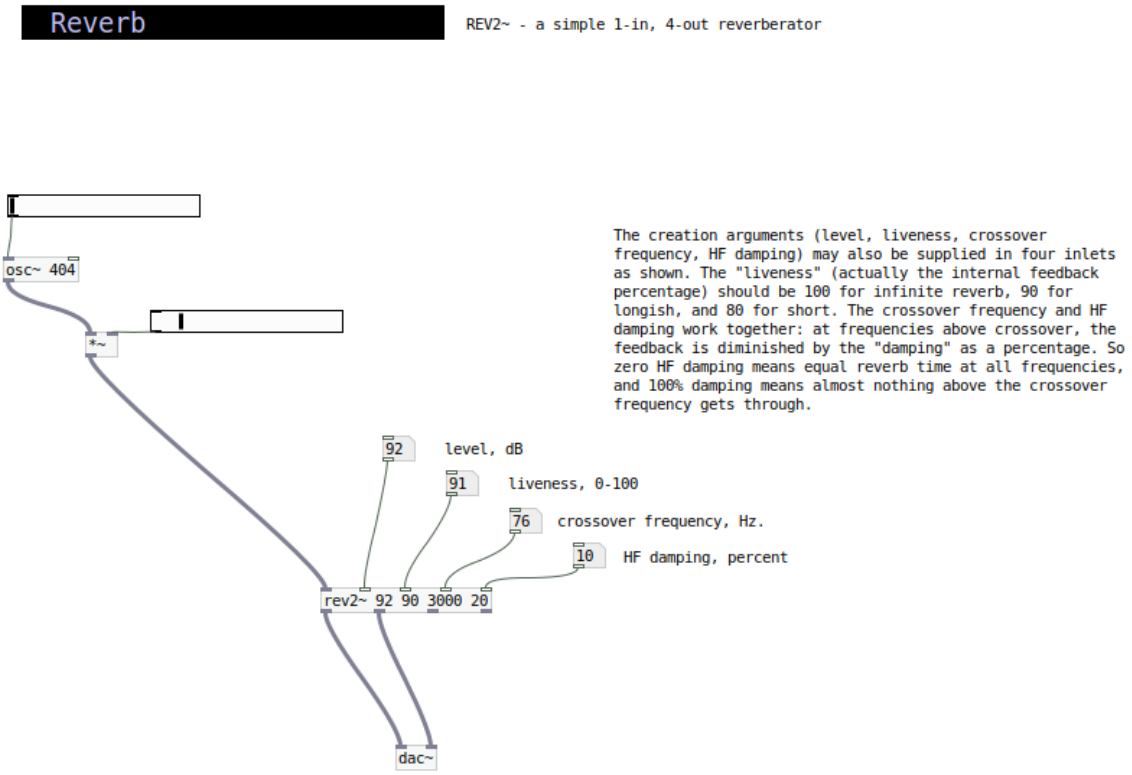
In this case the algorythm to quantize time, is as simple of use **[t b f] trigger bang float** which from a certain incoming value (in this case **[r timequantized]**), can be easyly sliced or quantized in proportions of time of the main clock with the initial sent **[s timequantized]**.
Therefore *Quantizations* can be controled just by the lilac Hradio Button.
In this case there is a couple of Hradios : on the left for quantized macro delays and on the right for quantized micro delays.

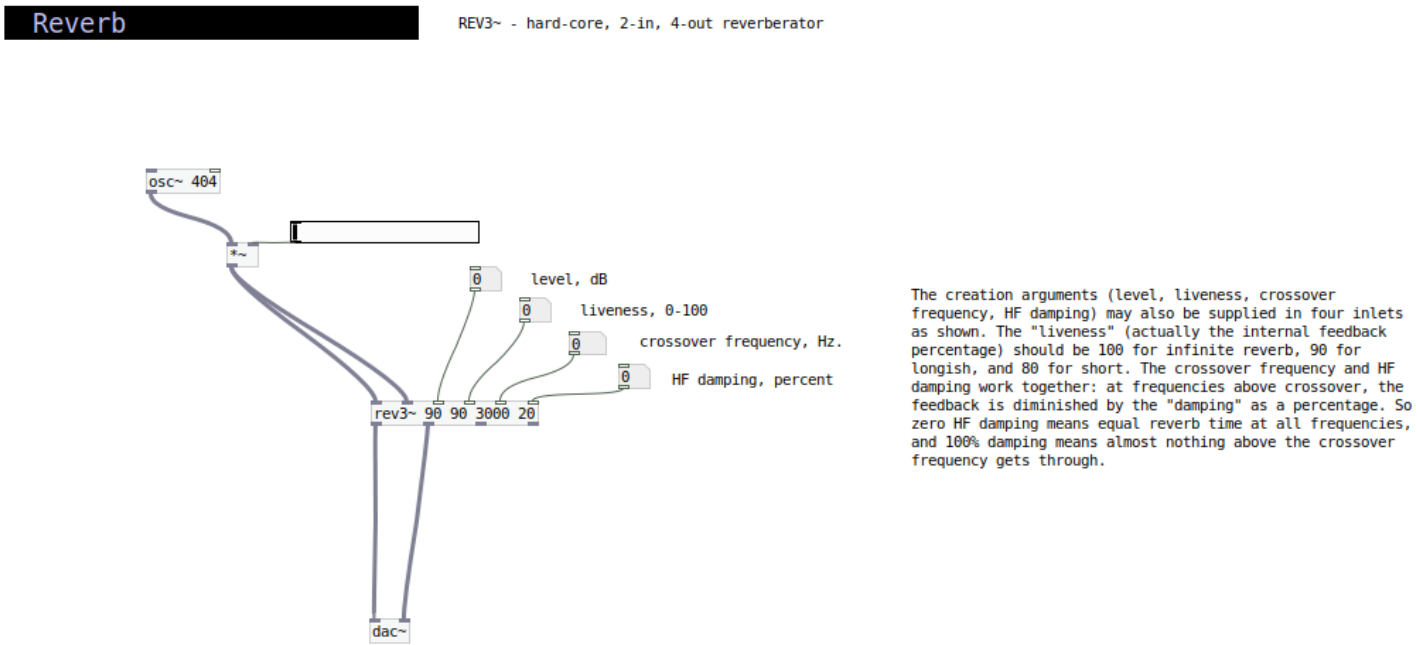


Filtering Signal

5.3 Reverb

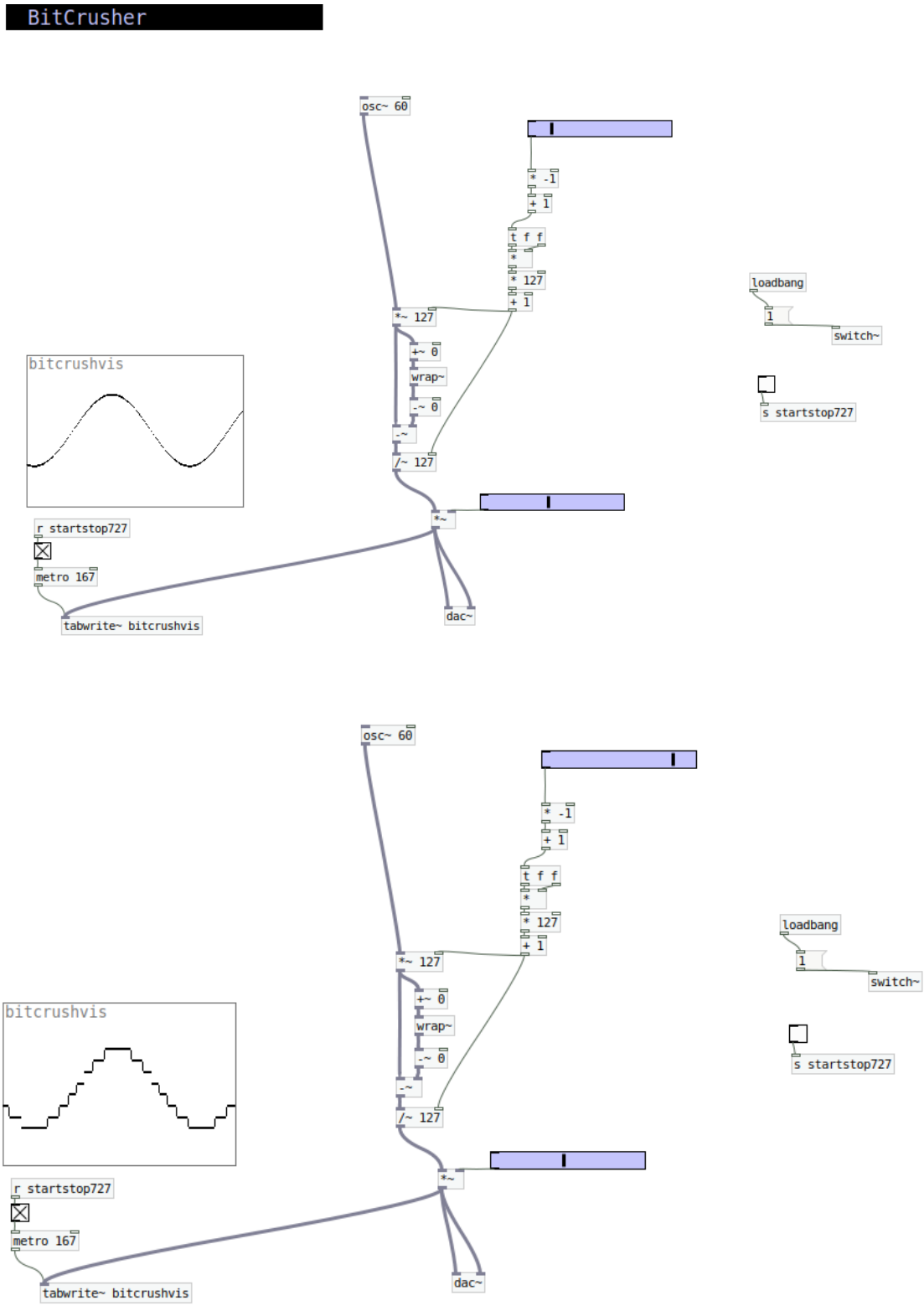
Like Delays**, Reverb** effect is an action that affects the incoming signal in the **time** domain, but specially in the **space** domain.
With this classic effect we can simulate different spaces from a tiny room, until a massive hall.
In Pd reverbs are pretty simplified objects, due to the expensive processor calculus of this kind of filters.
[rev2~] less Cpu expensive
[rev3~] more Cpu expensive





Filtering Signal
5.4 Distorsion

There are several kind of distorsion, but one of the most common is the **BitCrusher**, a type of distorsion that reduces the *bitdepth* of the running signal, altering the waveform itself.
The next example is an algorythm of bitdepthing.
For low values of the top lilac slider, incoming signal is not filtered
For high values of the top lilac slider, incoming signal is transformed into a much more ‘pixelated’ / squared waveform than the initial one.



Filtering Signal
5.5 WaveShapers

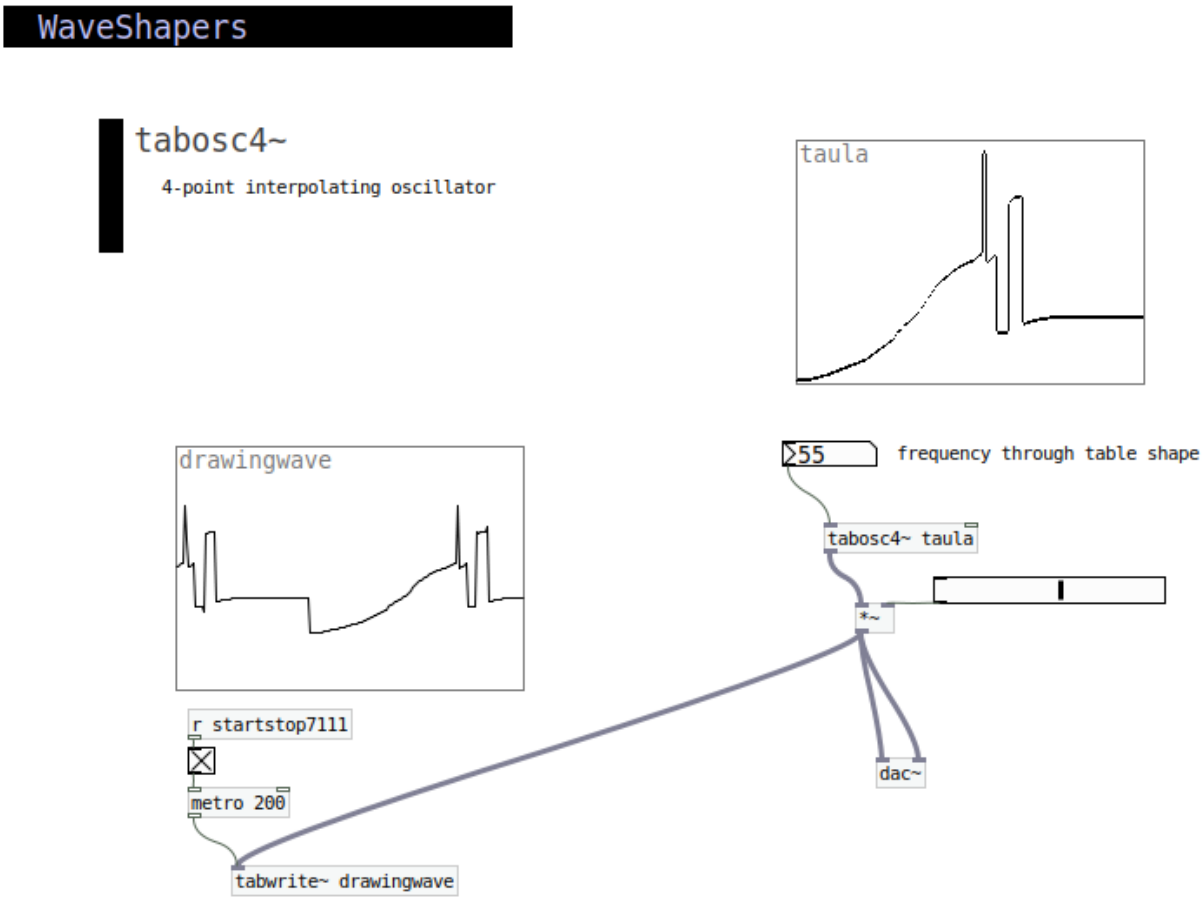
WaveShaping is an sculpted/extruded technique of an incoming signal.
According to a certain stored waveform's geometry, signal is processed with this shape.

In pd we can use different methods to build this filter.

tabosc4~

With this object associated to an array or memory (in this case ***taula***), an incoming value is processed as the lead frequency of an oscillator which shape or waveform is the one described in the array (***taula***).
In DSP terminology, [tabosc4~] is a traditional computer music style ***wavetable*** *lookup oscillator using 4-point polynomial interpolation.*

It's a nice technique to manage waveshaping with oscillators in a simple way.

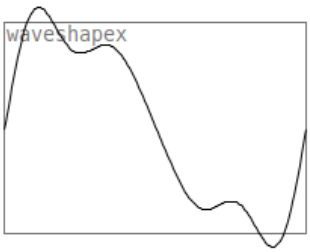


tabread4~

In DSP terminology, [tabread4~] is used to build samplers and other table lookup algorithms. The interpolation scheme is 4-point polynomial.

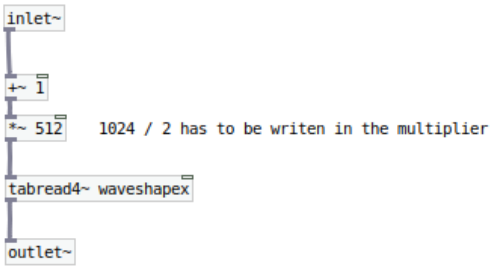
With this method **[tabread4~ array]** we can modelate in an easy way different shapes in the array, with trigonometric messages like sinesum, cosinesum among others.

With this method, window size **(X)** in samples has to be powered 2 (32,64,128,etc), that we can call with the appropiate messages.
[; array sinesum X string.of.values.that.describes.shape [
and in the muliplier of the signal featured in the examples [*~(X)/2]



1024 sample array

note Due to interpolation, size is increased in 3 units more in this case 1027



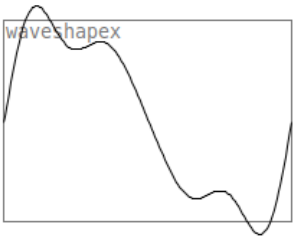
Notice that the more window size (X) will have the waveshaper, the more accurate or less glitched will be the resulting signal.
For example :

WaveShapers

tabread4~

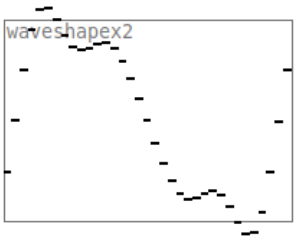
4-point-interpolating table lookup

arrays has to be powered 2 and after rewrite in the multiplier the size of the array



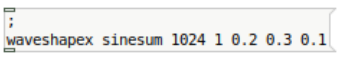
1024 sample array

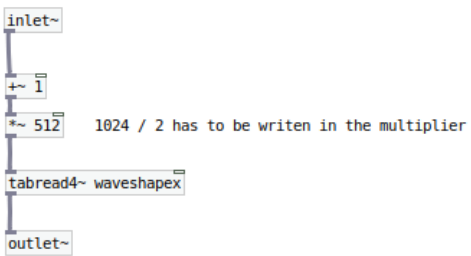
note Due to interpolation, size is increased in 3 units more in this case 1027

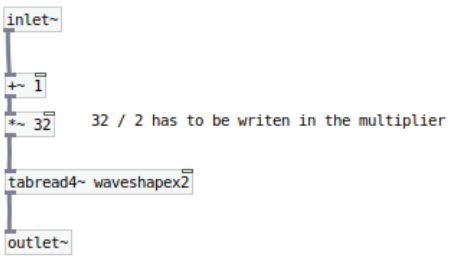


32 sample array

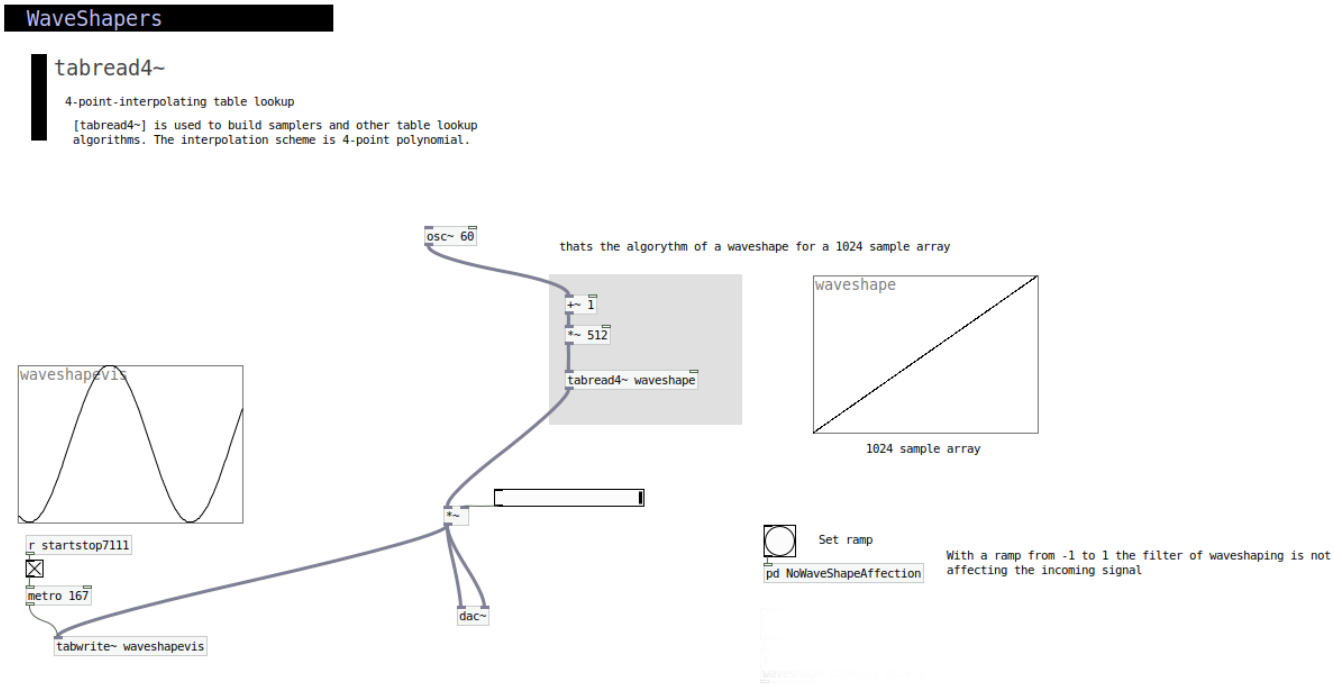
note to interpolation, size is increased in 3 units more in this case 35



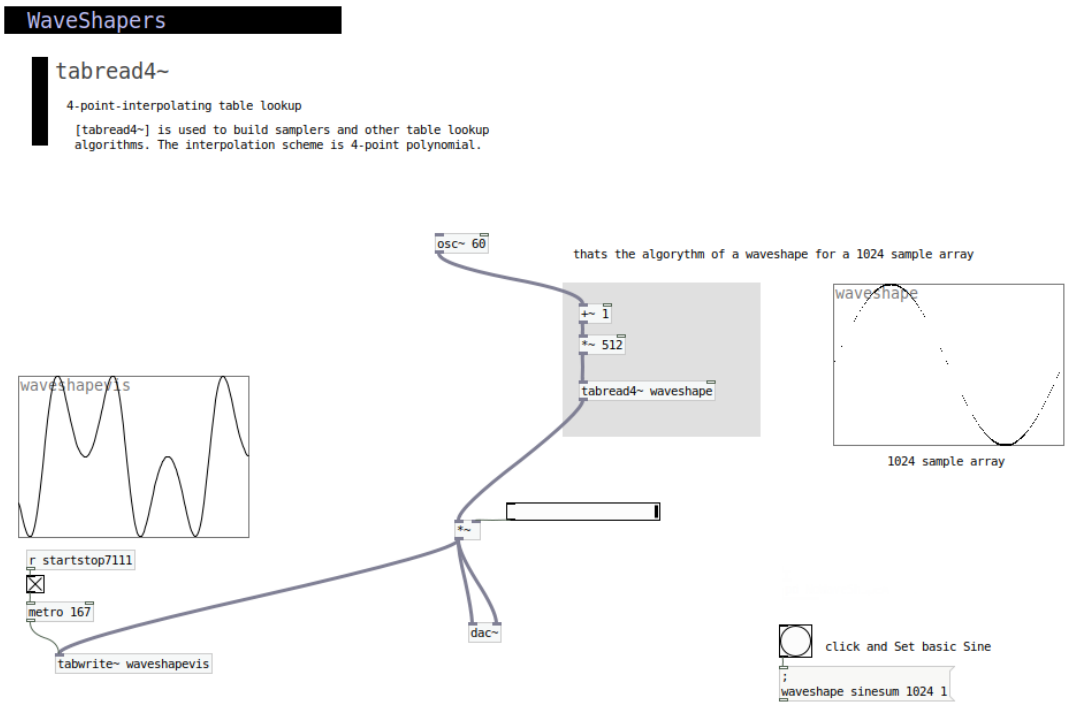




Following in this line of comments, lets see how behaves the array [in this case *waveshape*] depending on the written data on it.
If we have a simple ramp, incoming signal is not processed, so the result its gonna be like if there is no waveshaping filter at all. See this figure :



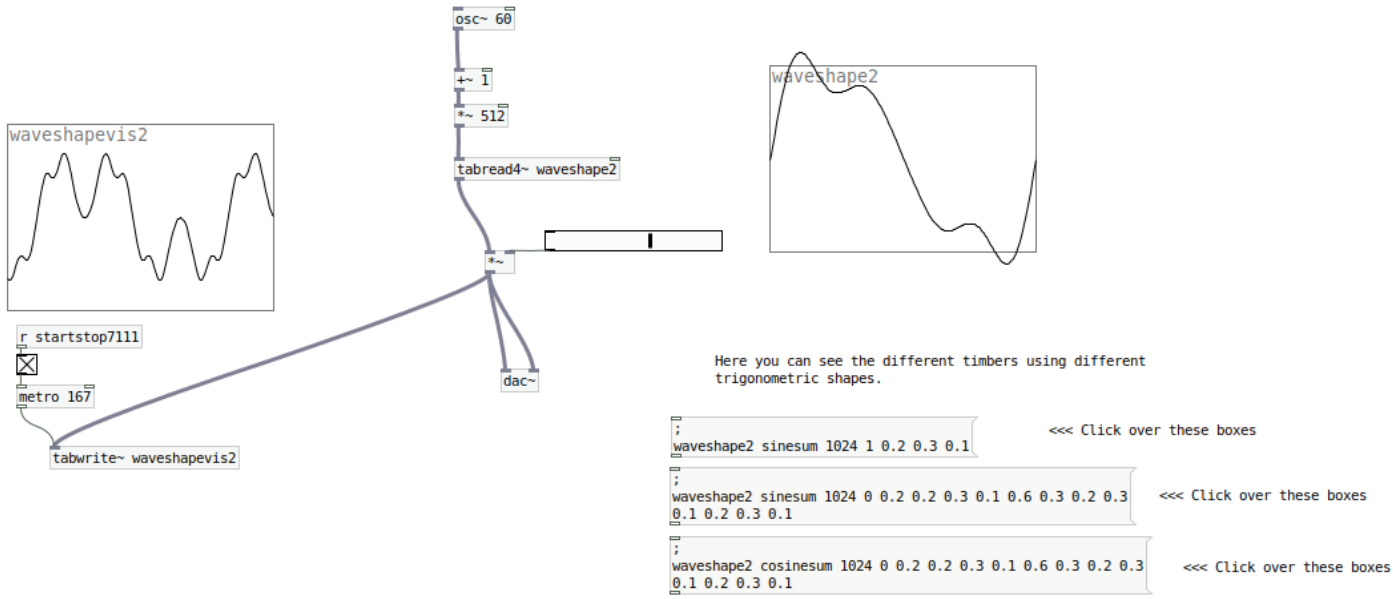
Otherwise If we have different shapes from the lineal ramp shown before, incoming signal is processed through the geometry of the draw shape, like if the previous ramp was the axis of the calculus.
In this example, a simple sinewave is producing a wooble wave like in visualization's array *waveshapevis* is featuring :



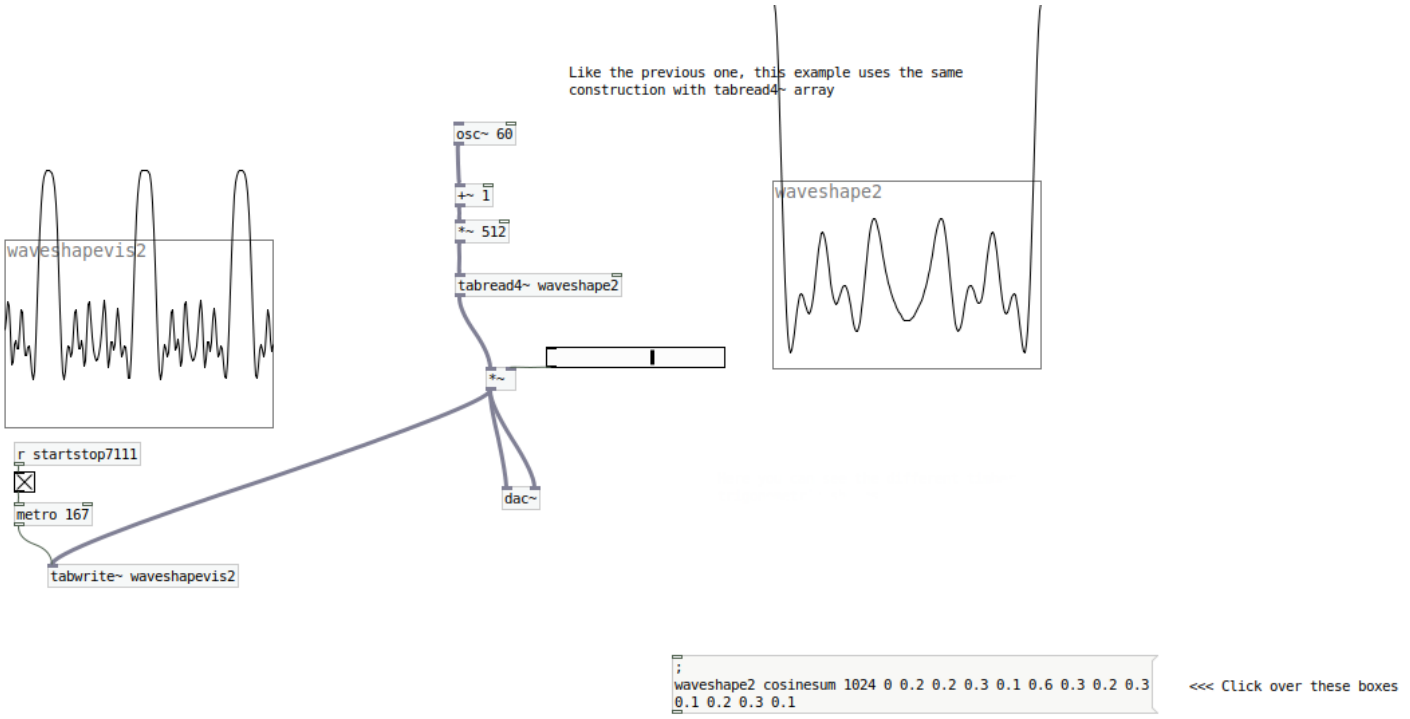
With this method we can easyly build different shapes, for example building with trigonometric messages : for example :

[; waveshape2 sinesum 1024 1 0.2 0.3 0.1[

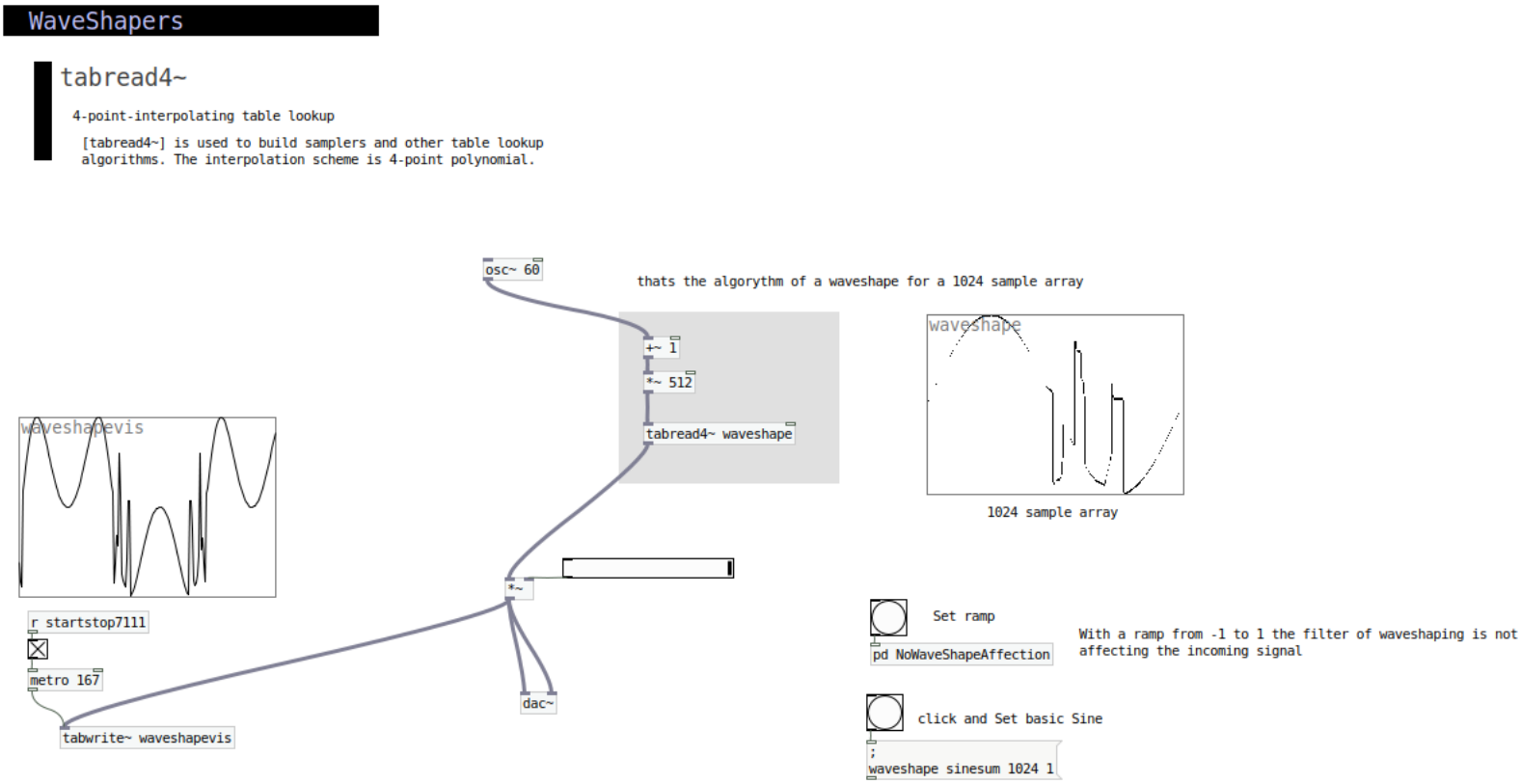
With sinesums we can produce similar effects to a compressor due to the fact that boosts incoming signal without clipping (in case that the stored shape is fit in the limits of the array (-1 to 1).



[; waveshape2 cosinesum 1024 0 0.2 0.2 0.3 0.1 0.6 0.3 0.2 0.3 0.1 0.2 0.3 0.1 [

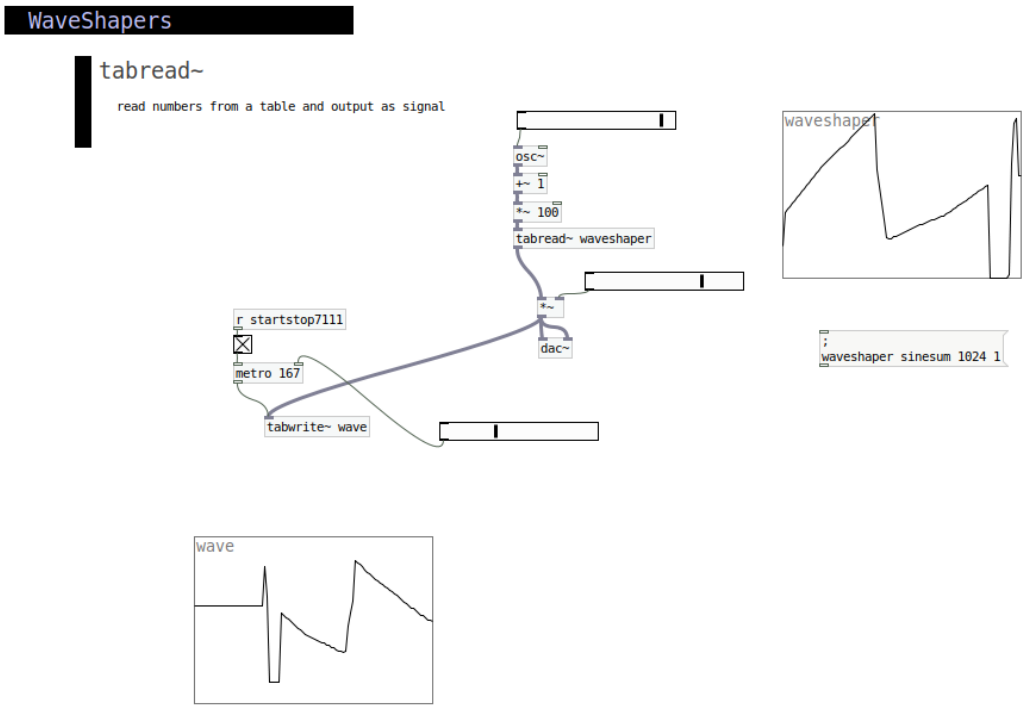


Another interesting feature with this method is to draw over the array with the mouse so we can modificate shapes with strange and non regular geometries :



tabread~

In DSP terminology this object read numbers from a table and output as signal.
Like in the previous its useful for drawing distorted shapes. Notice that the multiplier and the array size is 100 by default. You can change this in order to produce some more distorted effects.



Waveshaping RECAP :

May be you may think, that due to have several waveshape options and methods, which is better to use?

If you want to make noise registers may be **tabosc~** and **tabread~** are useful.

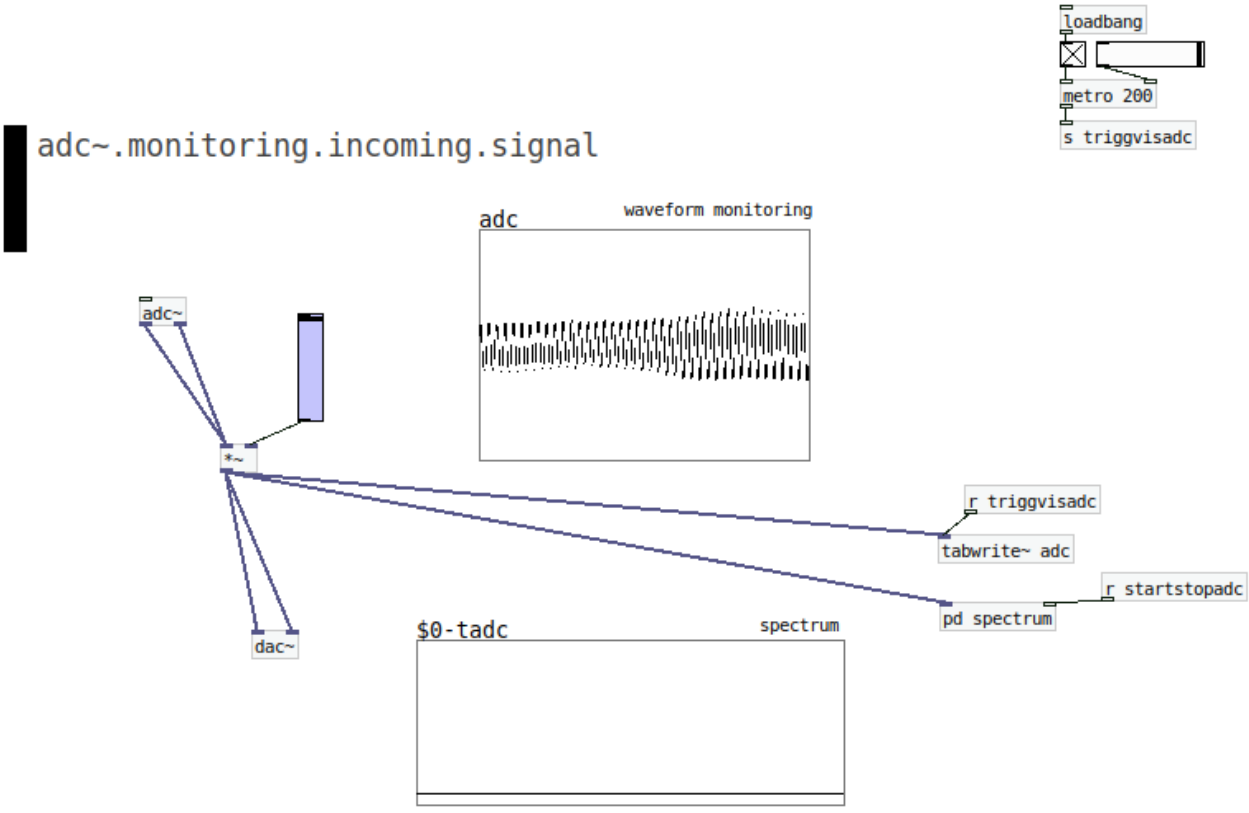
But if you wanna process your signal with more accuracy **tabread4~** is a very cool method because allows filtering signal both in a clean way and in a distorted / glitchy way depending on your performance needs.

adc~
6. playing with incoming signal

[adc~] *analog to digital conversion* is the object that introduces any available* line-in or mic, already set up in your sound card parameters. In laptops usually is the incoming signal of the in-built microphone, unless you load an external sound card with its incoming ports. In this case if you have an external soundcard with for example 4 mono inputs, those will be refered as **[adc~ 1][adc~ 2][adc~ 3][adc~ 4]**.

In LICH module we have a couple of incoming audio signal ports **(IN _L)** and **(IN _R)** that corresponds to **[adc~ 1]** and **[adc~ 2].**

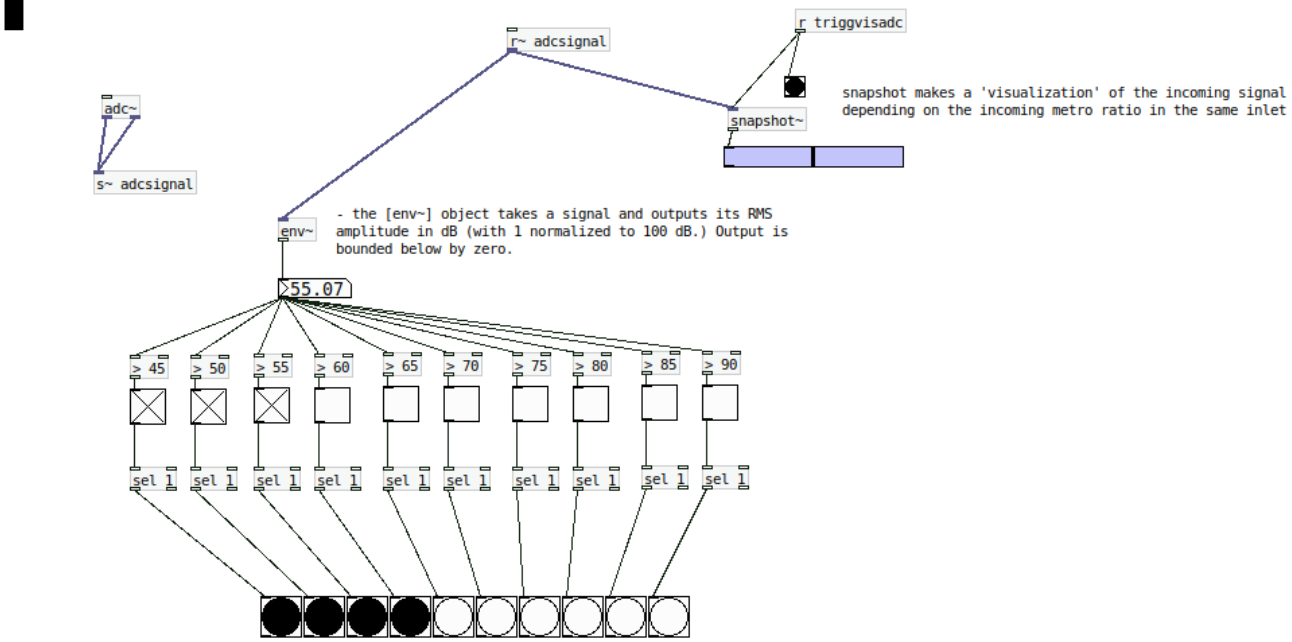
This piece of code features the incoming signal's render from the laptop's in-built mic.



With the object **[env~]** we can monitor the envelope of a certain signal in this case the incoming signal of the mic. The rate of analysed data will be very fast, but we can threshold it with different values in order to make some conditional tree of triggers according to the incoming signal's strenght.

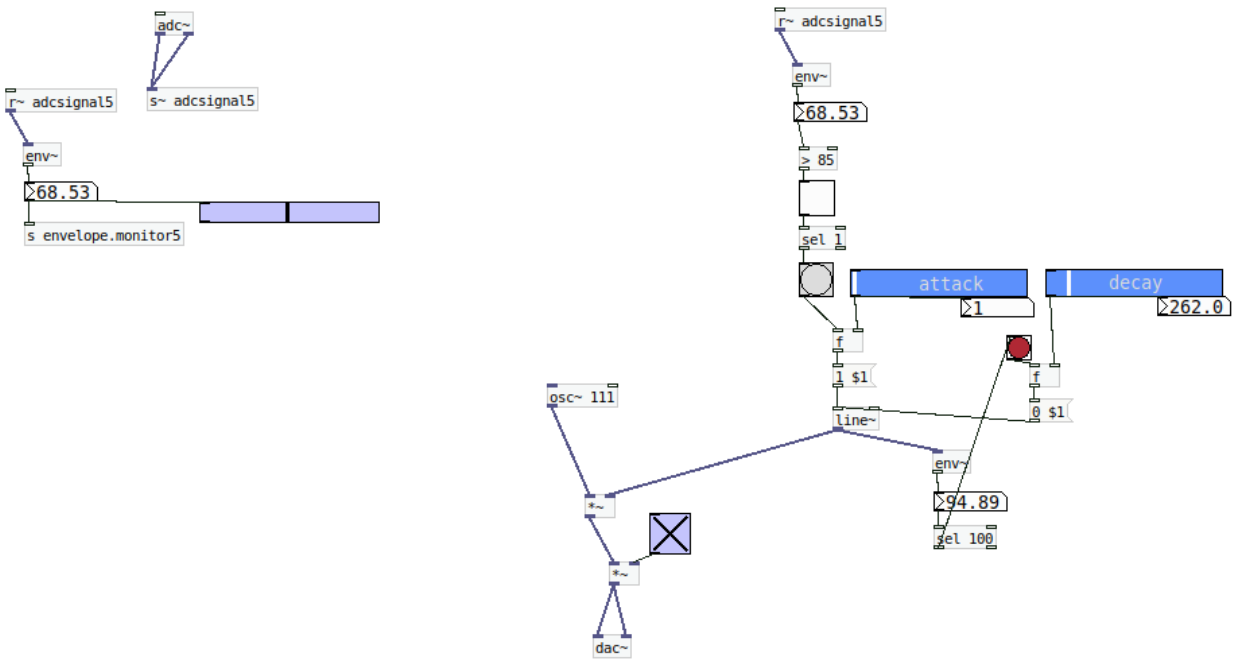
With the object **[snaphot~]** we also can monitor the envelope at a certain desired rate [with an incoming **[metro ms]** object, that maybe can match with our clock for other elements in the patch, that will bring us more sync effect.

adc~.envelope.thresholds



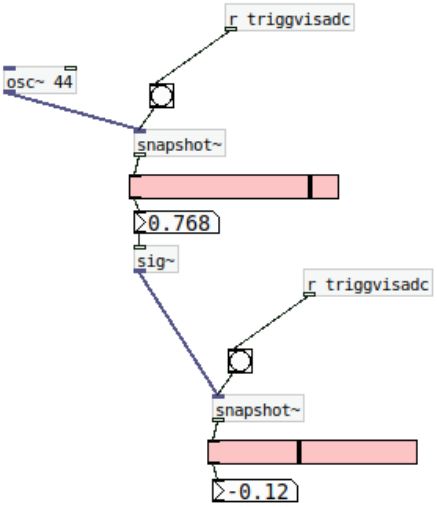
In this example, for higher values than 85 [> 85] detected in the [env~] analysis, it will trigg a bang that in this case is connected to an oscillator with attack and decay control, therefore a percussive sound.

adc~.envelope.controlling.triggers



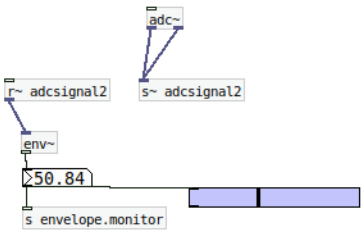
One way to translate signal to data is with the object **[snapshot~]** triggered with a [metro] object at the desired ratio.
On the contrary, one way to translate data to signal is with the object **[sig~].**

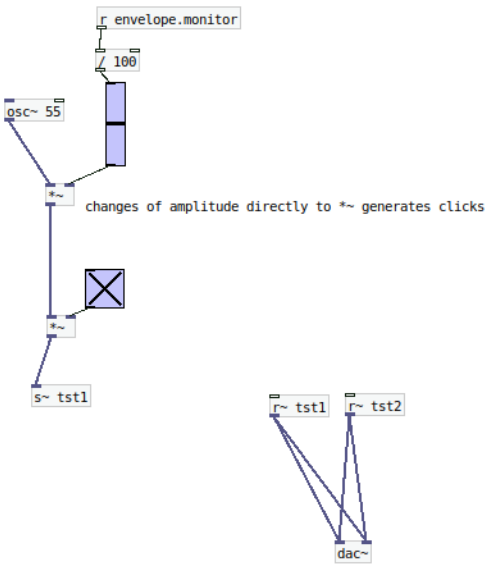
From.Signal.to.Data.and.vicecersa

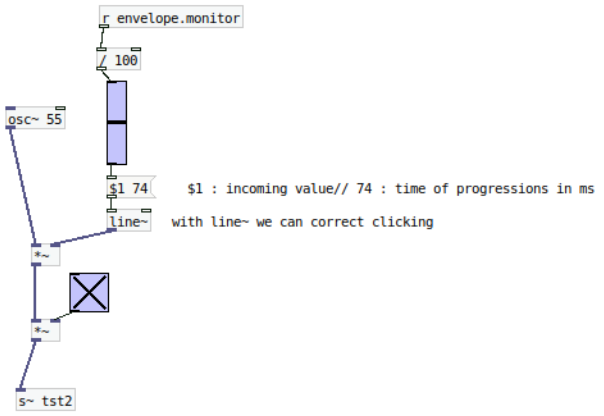


We can use envelopes to control generatively/automatedly different parameters. For example in this case the analyzed envelope data result is controlling the amplitude of a couple of oscillators.

adc~.envelope.controlling.amplitudes

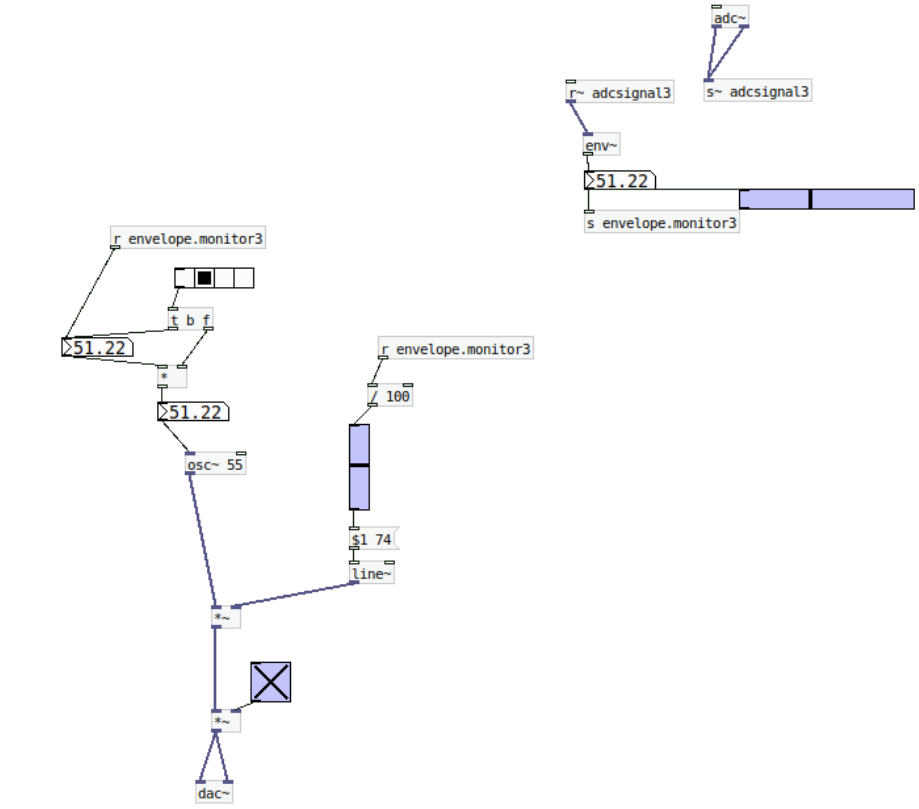






Or even the envelope can control the frequency and the amplitude of an oscillator, like in this example :

```
adc~.envelope.controlling.amplitudes.and.pitches
```



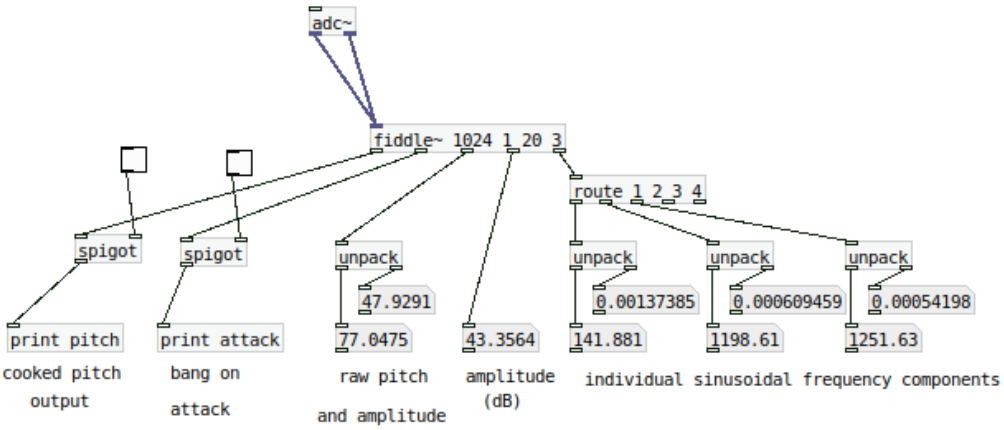
6.2. analyzing incoming signal

There are different objects useful for analyze a certain signal.
Anyways there a couple of methods that does not work in the rebeltech compiling process which are **[fiddle~]** and **[bonk~]**...but at least we can measure the incoming signal wtth **[env~]** as we already saw.

detection.functions.of.incoming.signal

fiddle~

this object does not correctly compile in Rebeltech LICH Browser Application

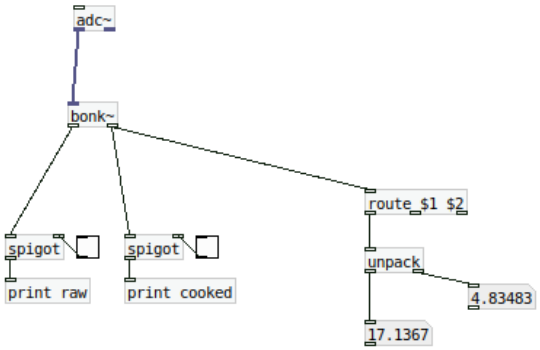


detection.functions.of.incoming.signal

bonk~

this object does not correctly compile in Rebeltech LICH Browser Application

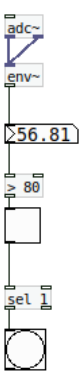
in order to grasp envelope detection try with [env~] and filter outcoming data as we saw before



detection.functions.of.incoming.signal

env~

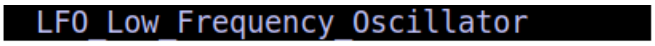
Remember to use
this method >>>>>
within your LICH Pd algorythms

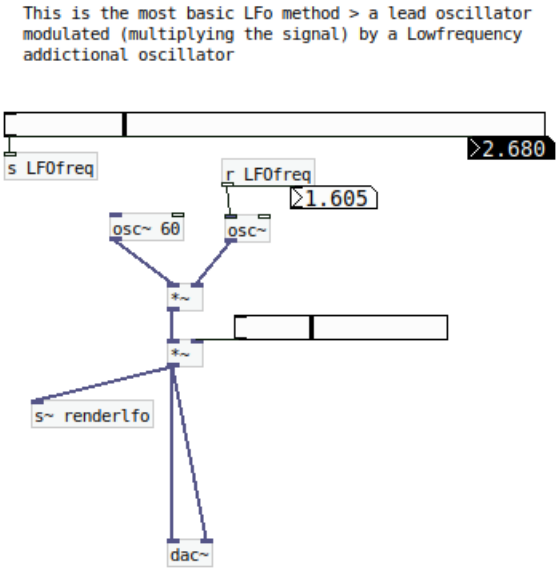


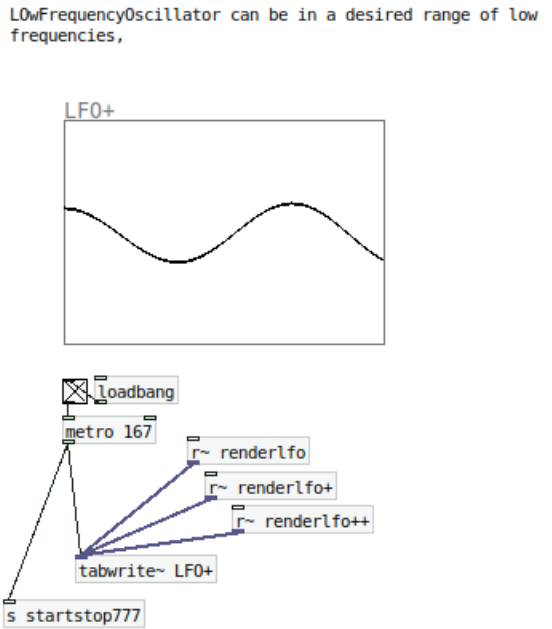
Envelopes & LFOs

7.1 LFO : Envelope Modulation by Oscillators

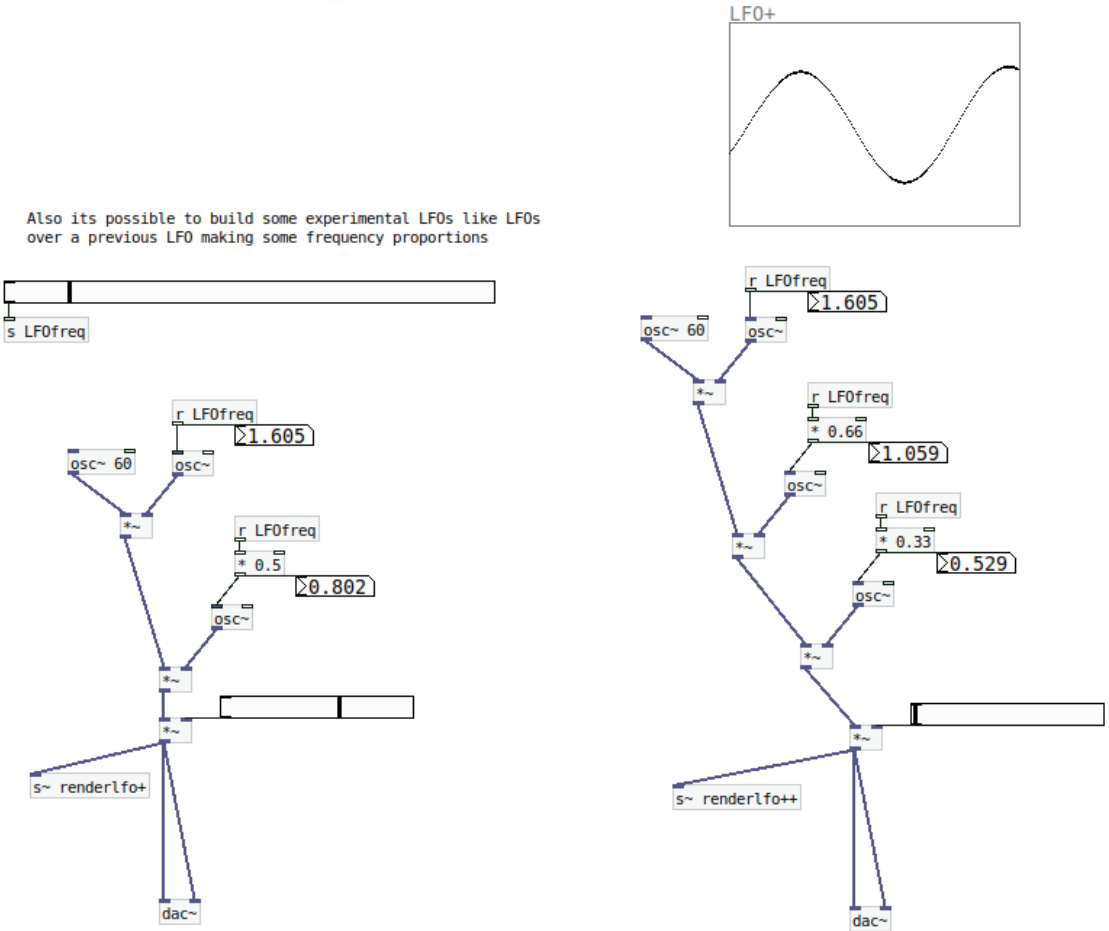
As you may know LFO is a very common and classic effect in electronic music.
It consists into the modulation between a couple of oscillators, one the lead frequency, and the other the LFO modulation rate with very low values of frequencies.
In pd the most basic method to build an LFO is multiplying the signal of two oscillators with very different frequency rates (the lead one and the modulator one).



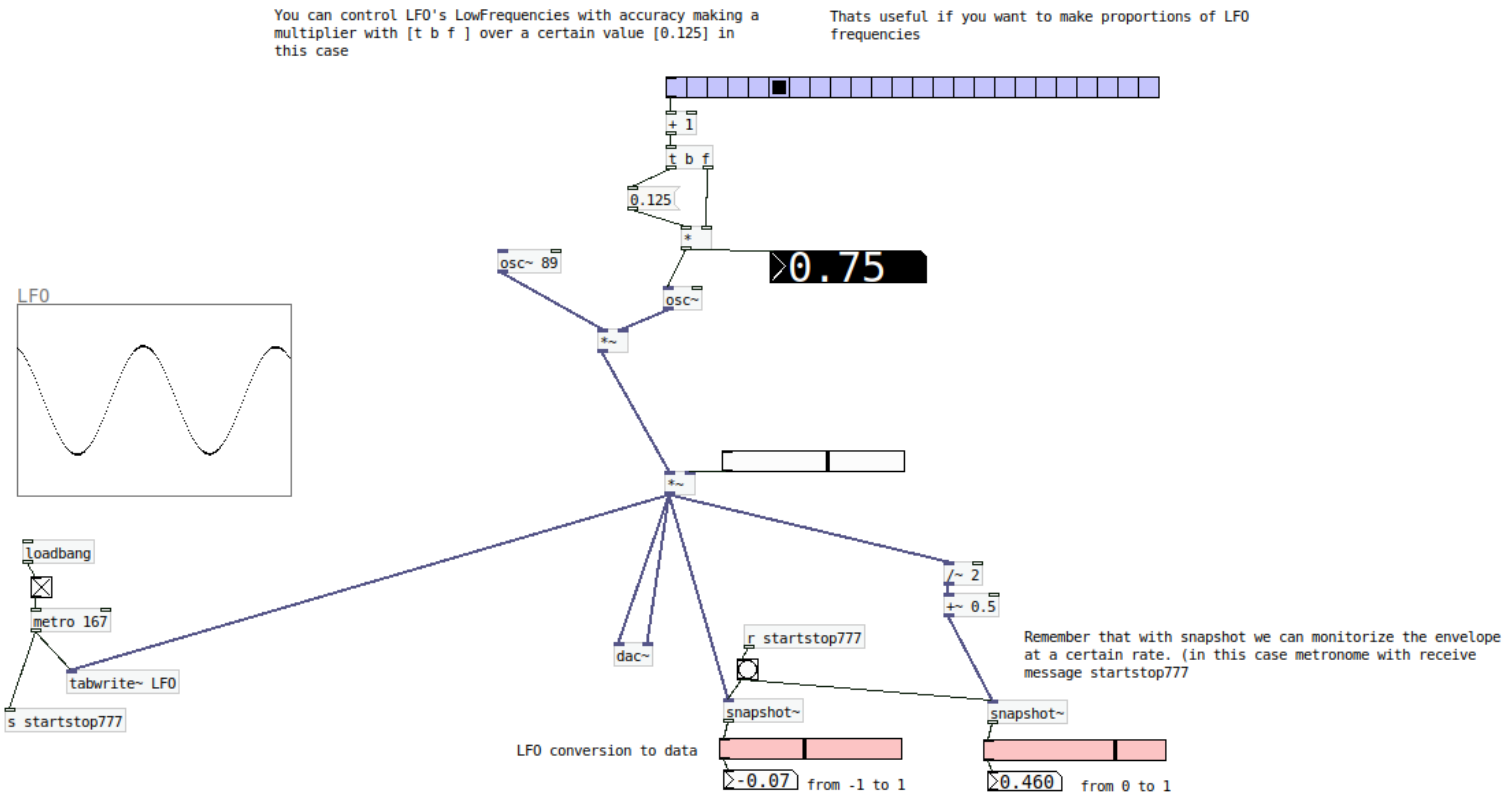




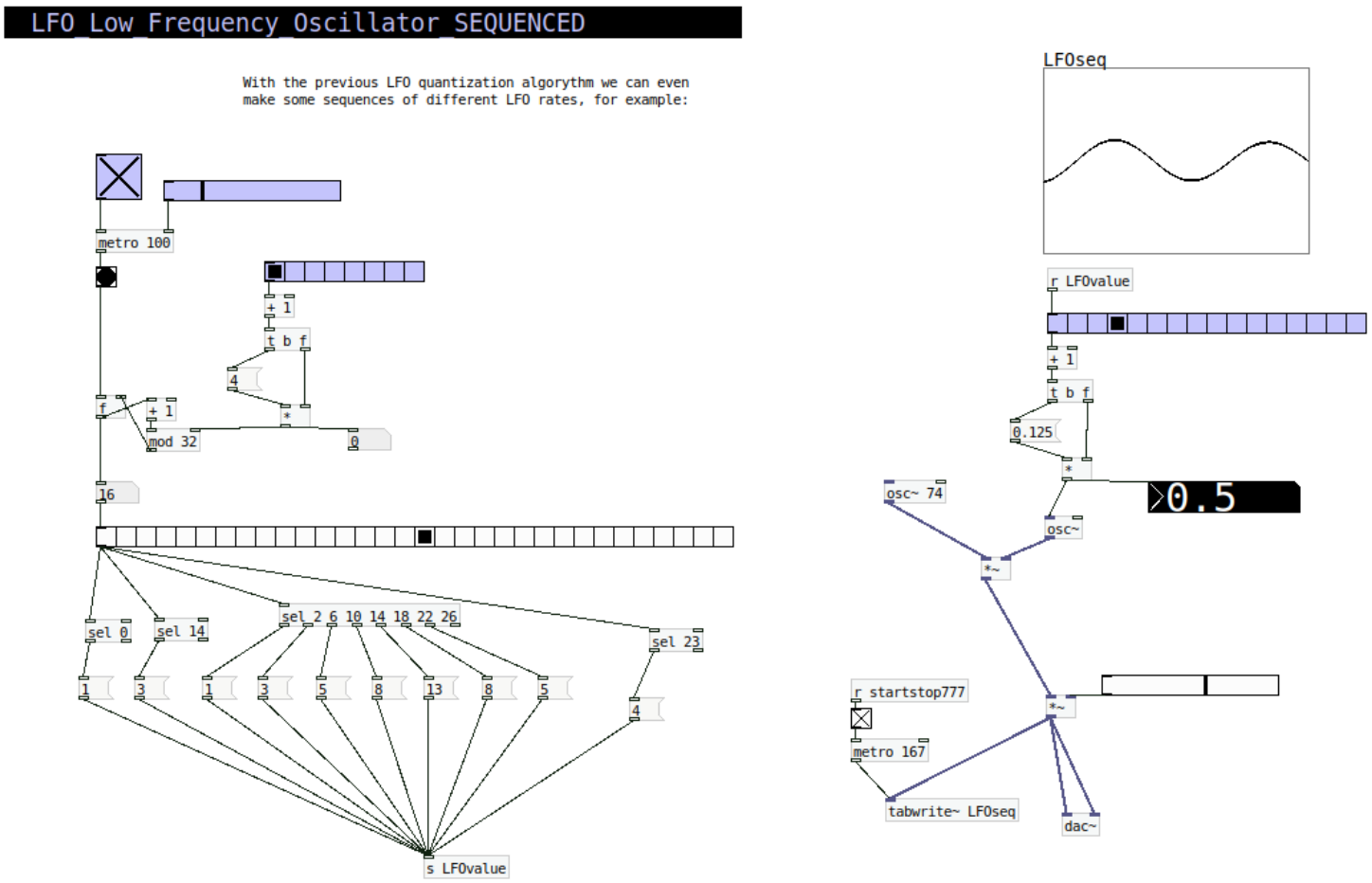
Also we can easyly build some more unconventional and experimental LFOs like put a chain of LFOs therefore an LFO over a previous LFO and so forth.



Or also to be more accurate in the LFO frequencies to tweak with proportions of a certain value (in this case **[0.125[**).
In this sense, quantizing LFO frequency rates like the previous one we can build some particular sequences



of LFO values



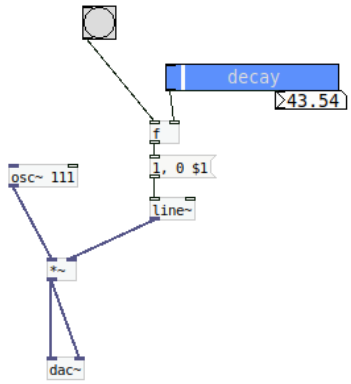
Envelopes
7.2 Playing with envelopes in percussive synths

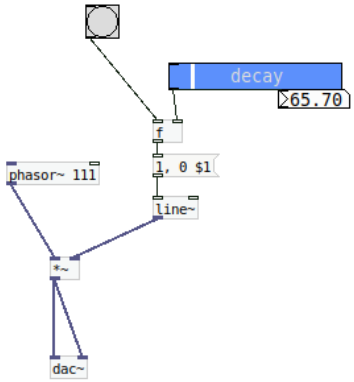
Synths are amazing instruments converting electricity into acoustic signal.
In digital world we would say that different kinds of data produces changes in the DSP extracting an acosutic signal.
In pd there are many kinds of types and techniques, but let's see how to build percussive synthetic sounds.
To produce them we have to control envelope as the classic AttackDecaySustainRelease structure, but even we can go in a much more simplified way, just only controlyng *decay* or *attack* + *decay*.

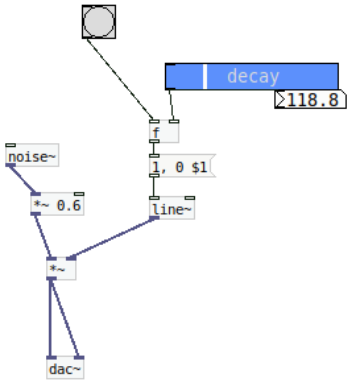
First we need signal generators that in pd -LICH oriented (vanilla), we have a couple of functions to do so, and already saw them :
[osc~] *sine oscillator*
[phasor~] *half sawtotth waveform oscillator*
[noise~] *white noise generator.*

With those objects we can produce signal that after we can control in its *decay* with the object **[line~]** that produces a ramp or progression between two values. In this case message **[1, 0 $1[** means that any time the trigger is activated, the envelope of the sound will go instantly to the maximum (1), and then goes to (0) (silence) in $1 miliseconds. As $1 is a variable, means that any value we are dinamically changing it will be cosidered as $1 (in this case with the blue slider).

percussive.synths

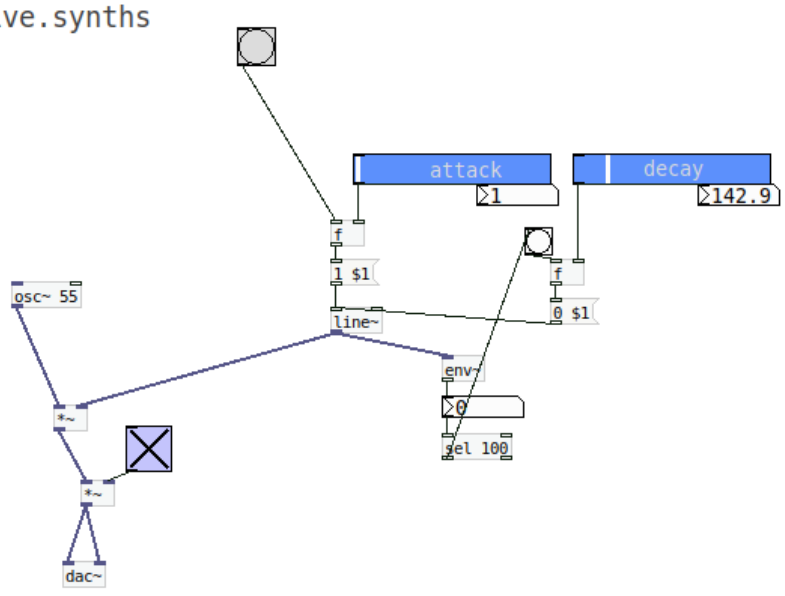






Also we can complex a bit the algorythm in the line~ section, introducing both values : one for *attack* and the other for *decay*.

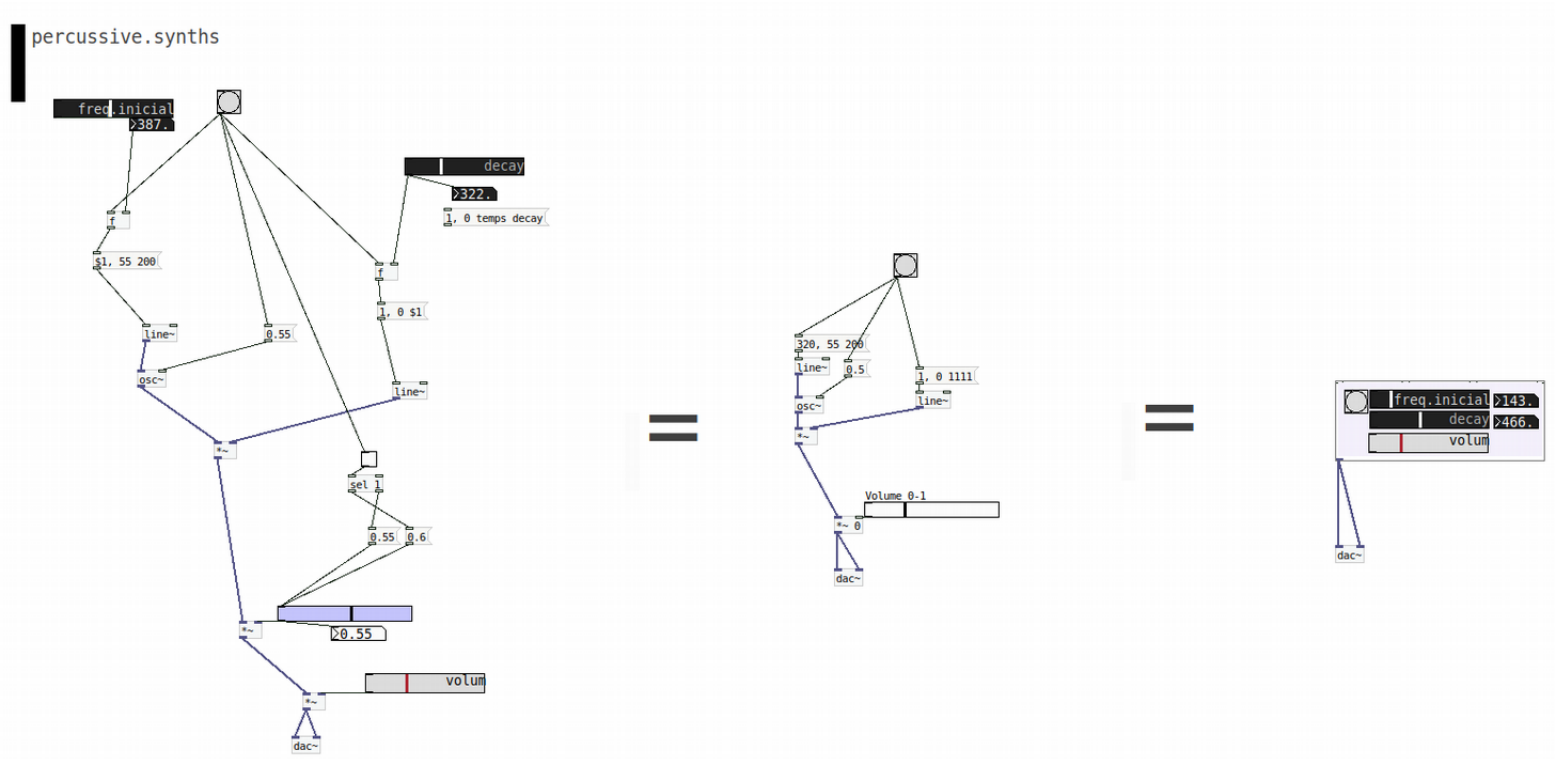
percussive.synths

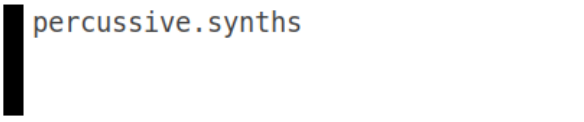


The next figure features the same algorythm in different compacted forms. On the left the whole algorythm, in the

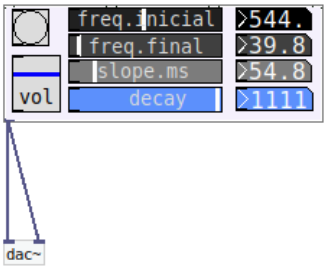
center a simplified one, and on the right an encapsulated method with graph [menú put > graph] that we saw in *2.2. Pd Packaging Algoryhtms*

Notice that line is not only applyed to envelope to control decay time, but also to slide in time the incoming frequency of the oscillator producing a *portamento* effect. In this case with the message [$1, 55 200[linked to **line~** in the left version, or [320, 55 200[in the center version.
In this last case, message **[320, 55 200[** indicates the start frequency (320hz), the target frequency (55hz) and the sliding time between both frequencies (200ms).





In this next figure >>>>> is shown a compacted version of a percussive sytnh where we can control *initial freq*, target or *final frequency*, sliding time *(slope)*, and *decay* time. Therefore a pretty compat and versatile module to trigg percussive sounds.



Sequencers
8.Time Machines

A Sequencer is one of the most essential tools in the electronic music production.
As you may know a sequencer is somekind of a time machine or time engine, because it builds narrative structures in time.

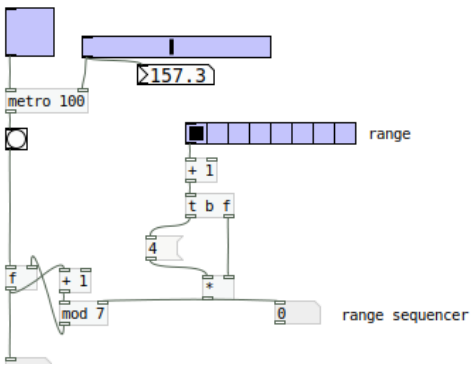
```
[ O ][    ][    ][ O ][    ][    ][    ][ O ]
[    ][ O ][    ][ O ][    ][ O ][    ][ O ]
[    ][    ][    ][    ][ O ][    ][    ][    ]
```

In the programming domain, this kind of structures can be pretty different, depending on the events that we want to create.

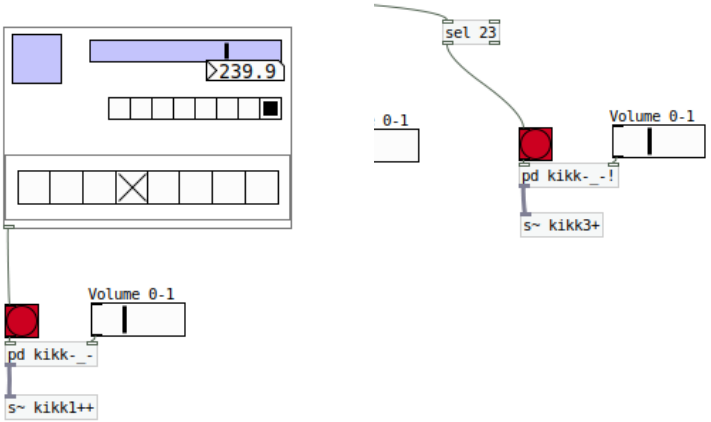
8.1 Linear Sequencers
8.1.1 Fixed

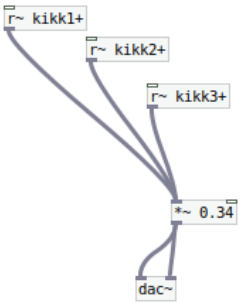
One of the most classic example is a fixed linear sequencer.
In this case building an algorythm with a loop section with **[mod]** and a conditional section with **[sel]**

Linear Sequencer : FIXED



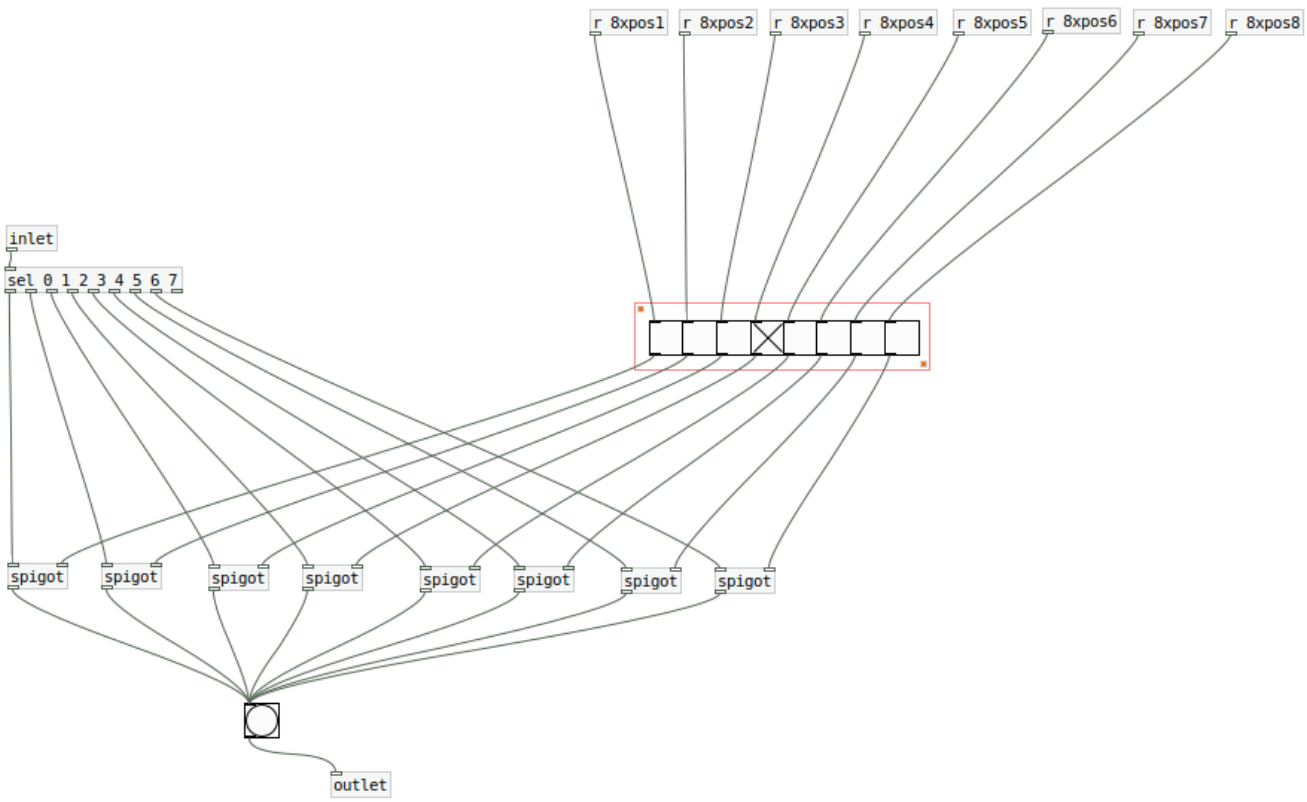
Linear Sequencer : 8steps





8.1.2 8 steps SEQ

This example is one of the most basic example of a swith sequencer with 8 steps, that que can swith with on / off any of those 8 steps.
Well need **[spigot]** object that opens or closes the gate* and also its receive messages for each step.
[s 8xpos1][r 8xpos1] [s 8xpos2][r 8xpos2] [s 8xpos3][r 8xpos3] .... and so forth
Notice that build this structures is somekind like a knit work, due to repeating a certain unit structure.
So if you want to build a 16 or 32 steps sequencer is simple although a meditative work)
of the incoming trigger that already has to be sequenced in postition with conditionals **[sel 0 1 2 3 4 5 6 7],*

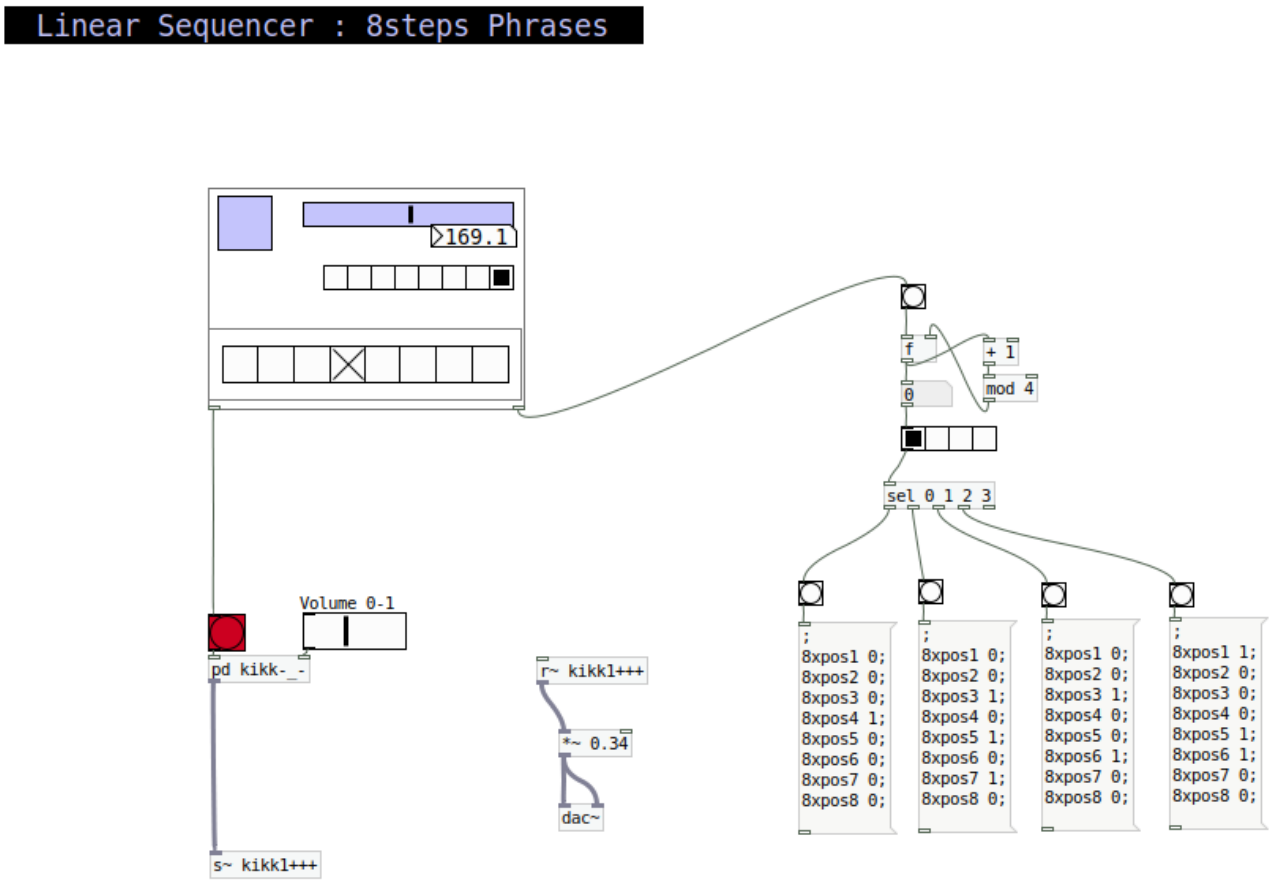


8.1.3 8 steps SEQ Phrases

This example creates a different stored pattern combination for every loop.
For do that we have to send and receive a 1 or 0 value in the ID message:
[s 8xpos1][r 8xpos1] [s 8xpos2][r 8xpos2] [s 8xpos3][r 8xpos3] .... and so forth
(the receive messages are featured in the previous image on the top)

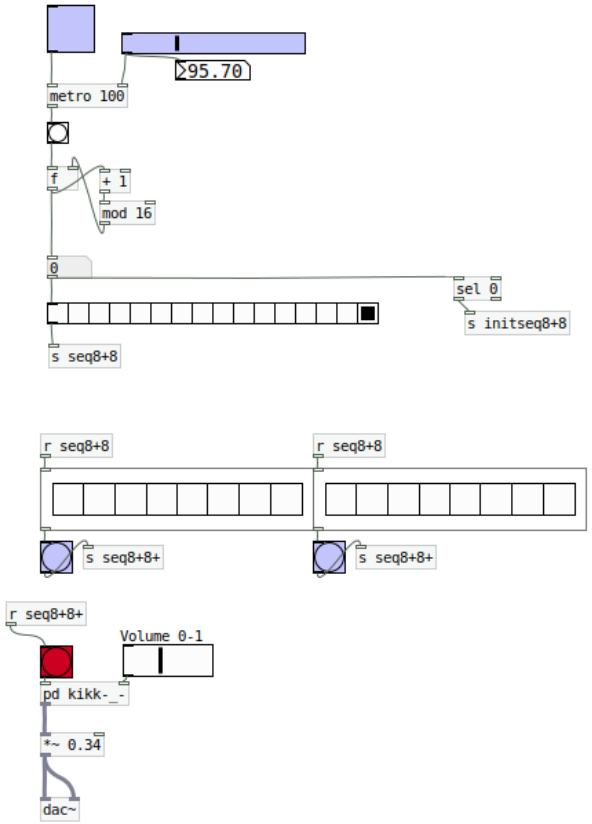
With the combined messages
;
8xpos1 1;
8xpos2 0;
8xpos3 0;
8xpos4 1;
8xpos5 0;
8xpos6 1;
8xpos7 0;
8xpos8 0;

is possible to set a particular combination of the whole 8 steps.
Therefore we can make a tree of patterns that are triggered every time the main loop starts.
In this case we have a tree of 4 different patterns but we can build some more complex structures in time in a pretty easy way.

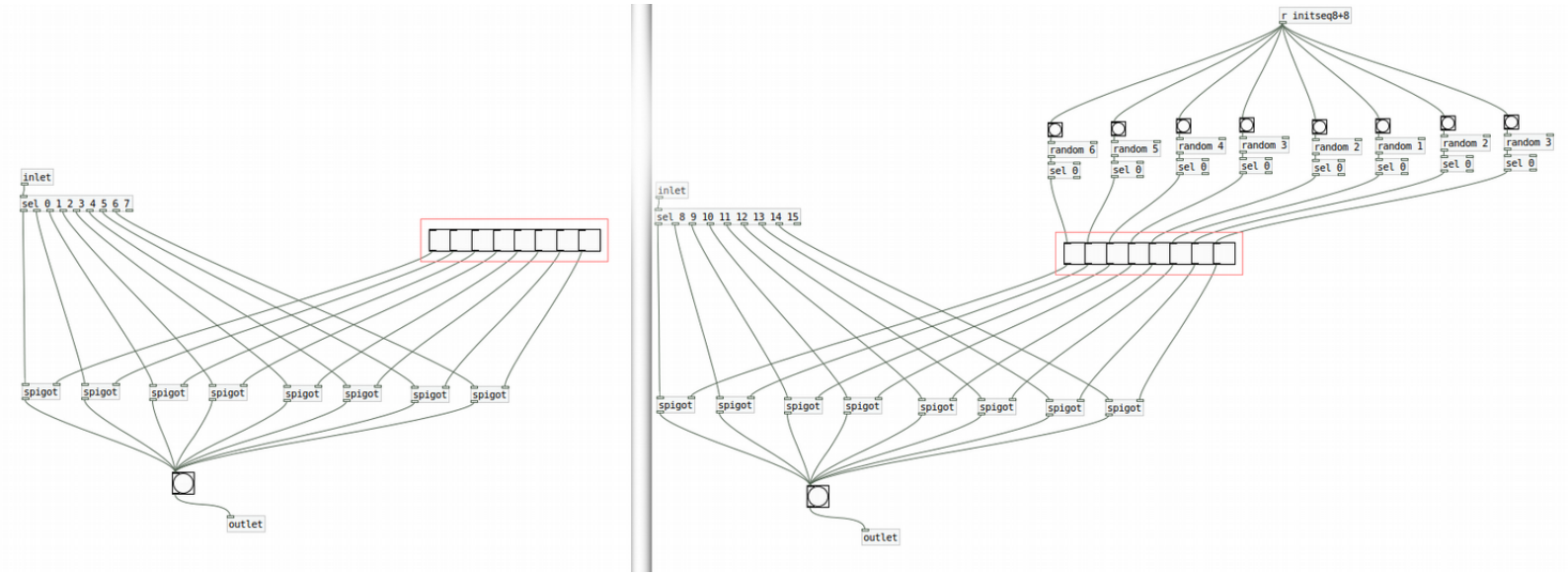


8.1.4 8 steps SEQ Fixed + 8 steps SEQ Random

Linear Sequencer : 8steps FIX + 8Steps random

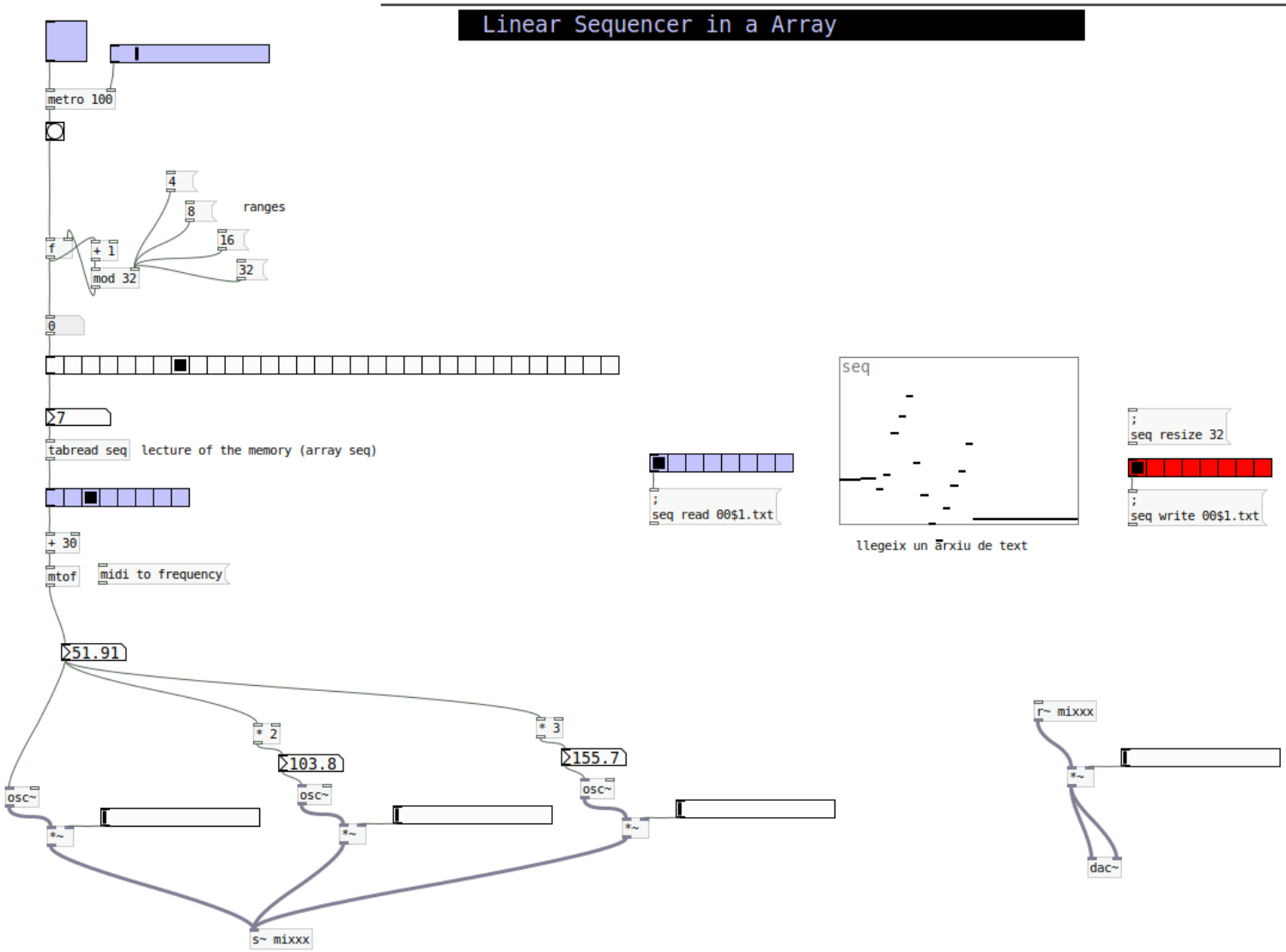


This example combines a 8step fixed sequencer where we can manually select the desired active step mixed up with a random 8 step sequencer with different proportions of random that can be tweaked or cancelled for a certain and desired randomization.



8.2 Sequencers in Arrays

Another method to define sequences is with Arrays.
In this case a looper is counting the content of an array with [tabread seq] that in this case trigs the storaged value as a main frequency of a synth with 3 superior harmonic oscillators.
Notice that the values of the array in this case are refered to a midi values. Therefore with the object **[mtof]** miditofrequency we can translate values to properly income in **[osc~]** objects

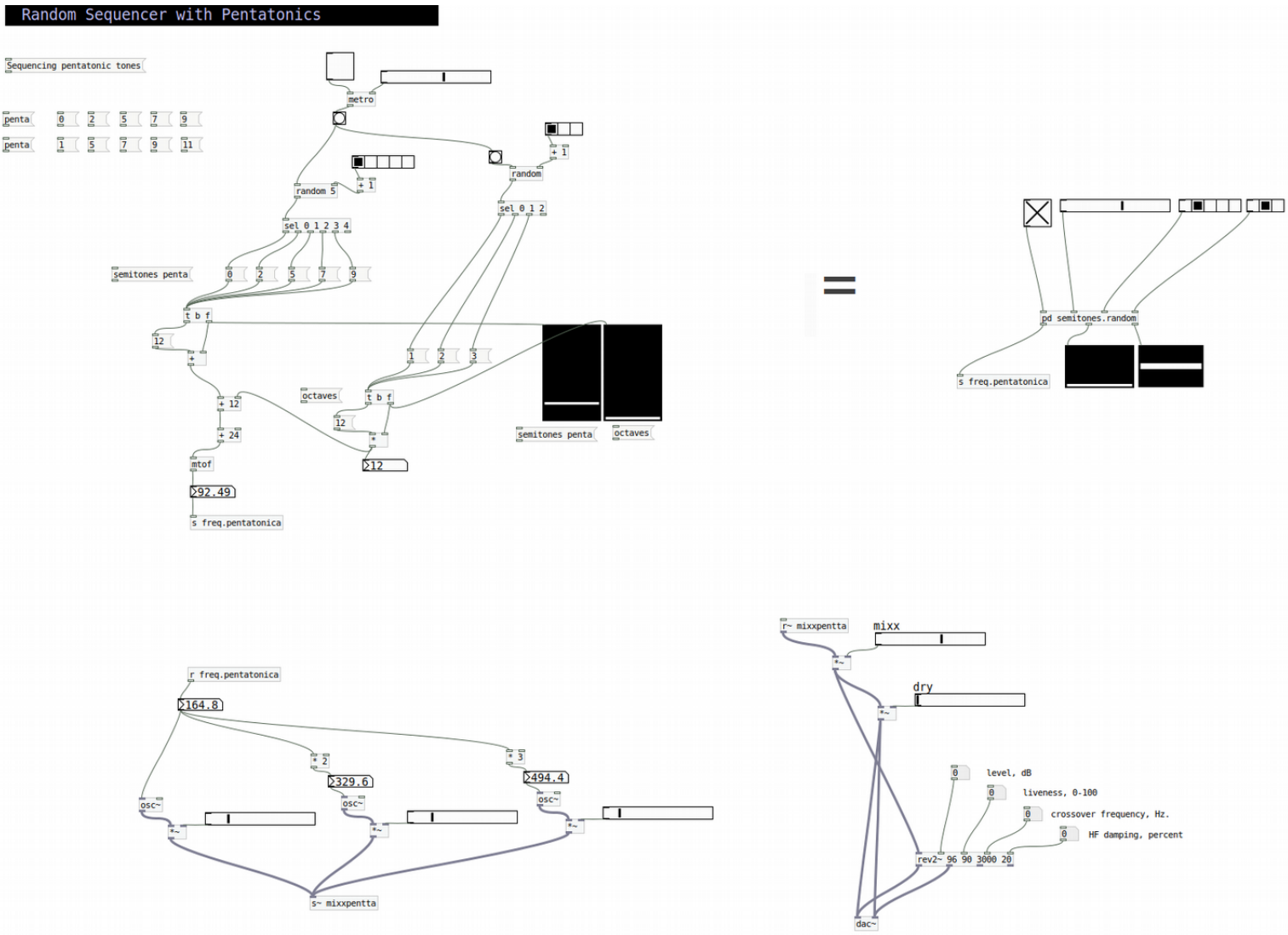


8.3 Random Sequencer with Pentatonics

When we are programming we can build nonconventional methods to build sequences.

For example, in this case a defined metronome is triggering a random value between those values that corresponds to a pentatonic scale combination. In the code the semitones relation **[0 [[2 [[5 [[7 [[9 [**

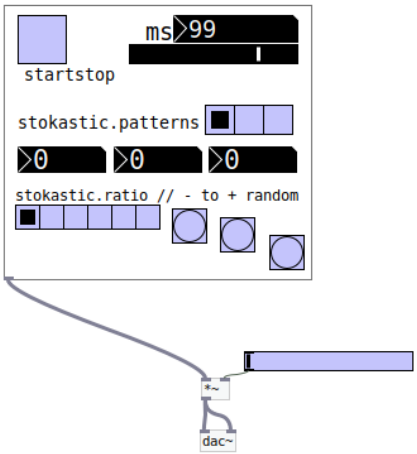
Usually playing with random is not a straight forward task for nice sonic designs, but in this case we have the advantage that between pentatonic tones every tone matches ‘harmonic’ with the others. Therefore any random combination between those semitones will be ‘audible-comfortable’.



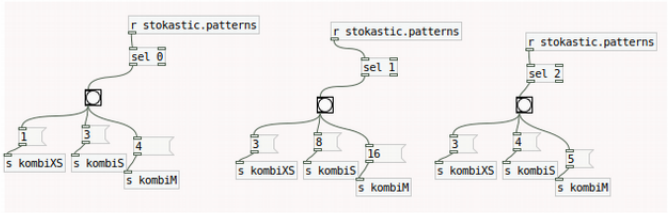
8.4 Stokastic & Probability Sequencers

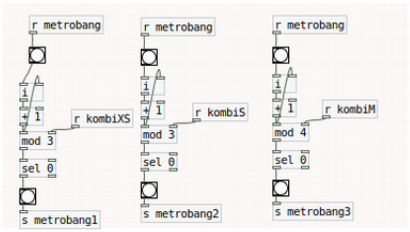
Probabilistic (Stokastik) Sequencer

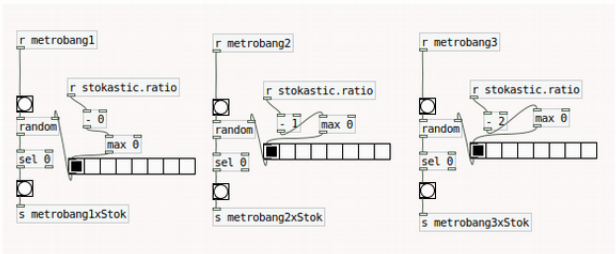
Another non conventional method, borrowed from Xenakis researches decades ago in probabilistic sequencers / synths, is this tiny example of 3 triggers (with 3 embed percussive sytnhs) that can be both triggered in 3 types of combinations and amount of stokastic ratio (from less random to more random probabilities that triggers are activated or not.)

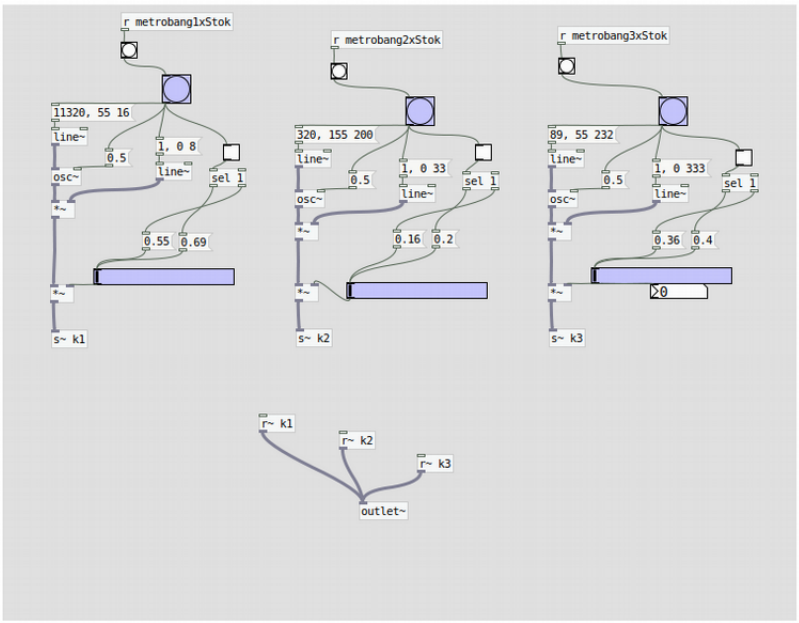


Here the internal code with all probability operations.





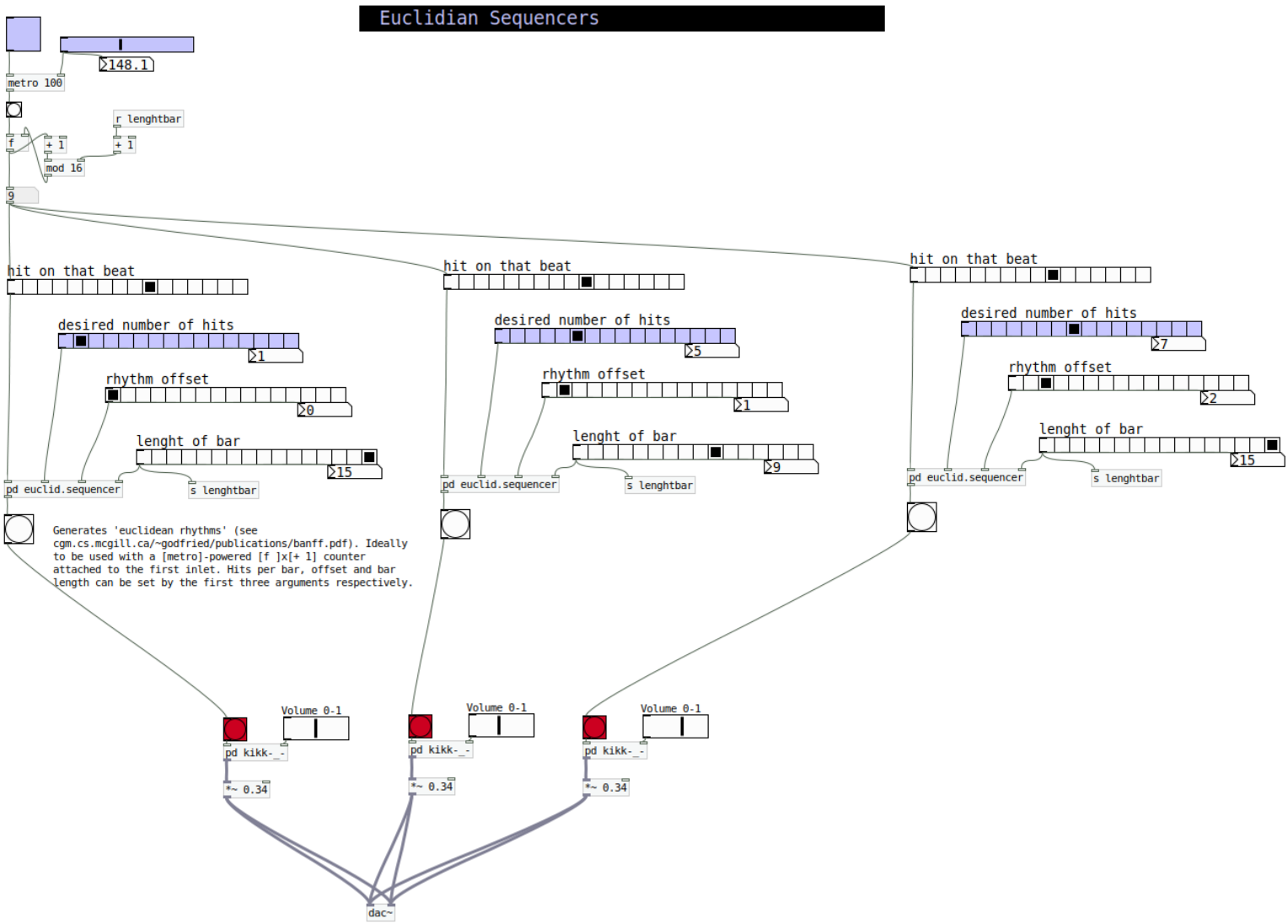




8.5 Polyrhythms : Euclidian Sequencers

A very interesting method for build polyryhtms is using Euclidian Sequencers.
As a reminder, Euclidean rhythms have their roots in Greek mathematician Euclid's algorithm and involve using the greatest common divisor of two numbers to place hits in a sequence as evenly as possible across a set timing division.

This technique allows us to build patterned and organic sequences due to the fact to being manage different ranges of sequences 'geometrically'. Therefore unusual beat combinations like 5/4 7/8 8/9 can be easyly combined as well with the most common static patterns 4/4 2/4 etc, in order to build unconventional but interesting beats.

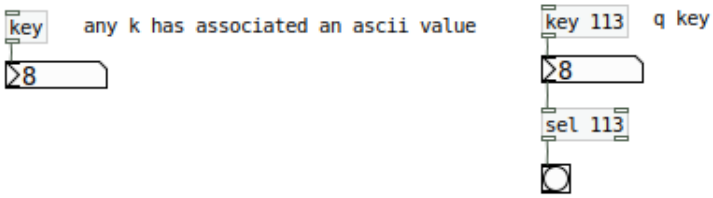


HIDs
10.Human Interface Devices

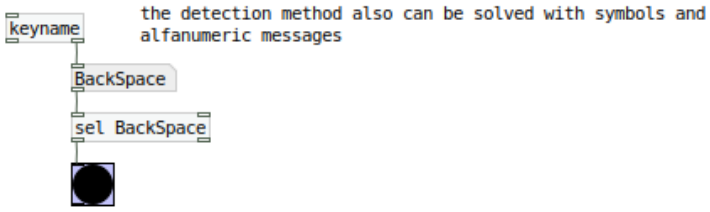
Useful functions and methods for Sonic Interaction

10.1 keyboard control

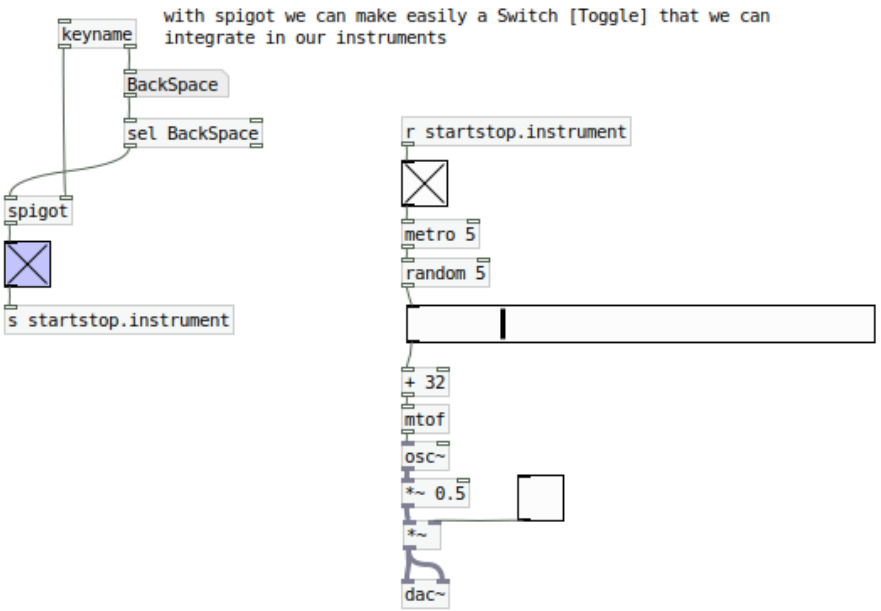
In pd we can easily manage laptop's keyboard as a triggers or switches.
We can use the object **[key]** which corresponds to a numeric value codified in Ascii. Every key has a certain number or alphanumeric description. In the first case we are using ***[key asciinumber]***.



In the second we are using **[keyname]** attached to a symbol.
Adding a [sel] with the incoming specific number or symbol we want to control we can easily have a trigger.



Also with spigot we can transform the simple trigger into a switch. See on the right >>



This is usefull to swith on/off certain algorythms we want to launch in time, like the most on the right section in this image >>>>

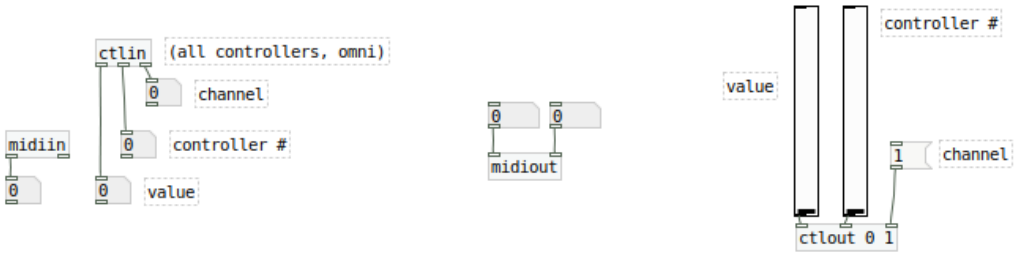
Another feature using keyboard is [keyup]. Thats not so efficient for triggers, but maybe for some special functions we need to control as soon as certain key is released can be ok.

Midi

11 Midi controllers

MIDI protocol is obviously integered in Pd.
Usually is used for Midi in messages from a midi controller, but also can be used in both directions sending also midi messages to another devices.

For input methods we have **[ctlin]** which extracts three outlets : the current midi value, the controller ID of a certain key knob or slider, and the channel (default 1).



Also this method can be written with the syntax **[ctlin ID 1]** that directly extracts the current midi value in its outlet (check next example).

For output methods we have **[ctlout]** which in an opposite way as ctlin integers three inlets : the current midi value, the controller ID of a certain key knob or slider, and the channel (default 1).

Note that to manage different midi IO devices those have to be set up in the preferences menú (menú Edit / Preferences).

As an example in this tutorial we have the midi mapping for a korg nanokontrol 2 device.
In case you have onther controller is very useful to make a mapping patch of it, in order to use with several instruments we want to develop.

Ex korg nanokontrol2 mapping