A Quick Guide to Rampcode

Brief reference by Gabochi

```
sin($v1/($v1/1000%
($v1/3000%4+1)+sin
($v1/2000%3))*sin(
$v1%1000/1000,
$v1/1000%7*3))*(1-
$v1)*(sin($v1%3200
```

Table of Contents

1. Rampcode....3
    2. The ramp....3
3. Sines....4
4. Sequences....5
    4.1. Counters....5
    4.2. Pseudo randomness....6
5. Control....7
    5.1. Amplitude....7
    5.2. Release....8
    5.3. Measure....9
    5.4. Scales....10
    5.5. Frequency....11
    5.6. Trigger....12
    5.7. Pulse....13
6. Distortion....14
    6.1. Clip....14
    6.2. Degrade....15

1. Rampcode

Rampcode is a synthesis system or technique inspired by bytebeat. It's about creating an artistic piece from the less possible elements: a single function with a single variable. I use it for live coding music through Pure Data but it can work for other purposes and adapted to other languages. Let's say you have this funcion: f(x) = x +2. If you evaluate it when x equals 1, it will return a 3 as output. Now imagine that the function is evaluated lots of times per second and that the output goes directly to the speakers and it sounds as music. That's all. I will assume that you already checked the patch and the setup documentation so I can focus on the expressions.

The more you know about digital sound processing, synthesis and expressions, the more you'll understand this guide but nothing stops you to start from here and learn about these things now, in the process. Also, this document is way beyond covering all the possibilities of rampcoding, its goal is to show a briefing of the basic aspects that work for me at this moment.

2. The ramp

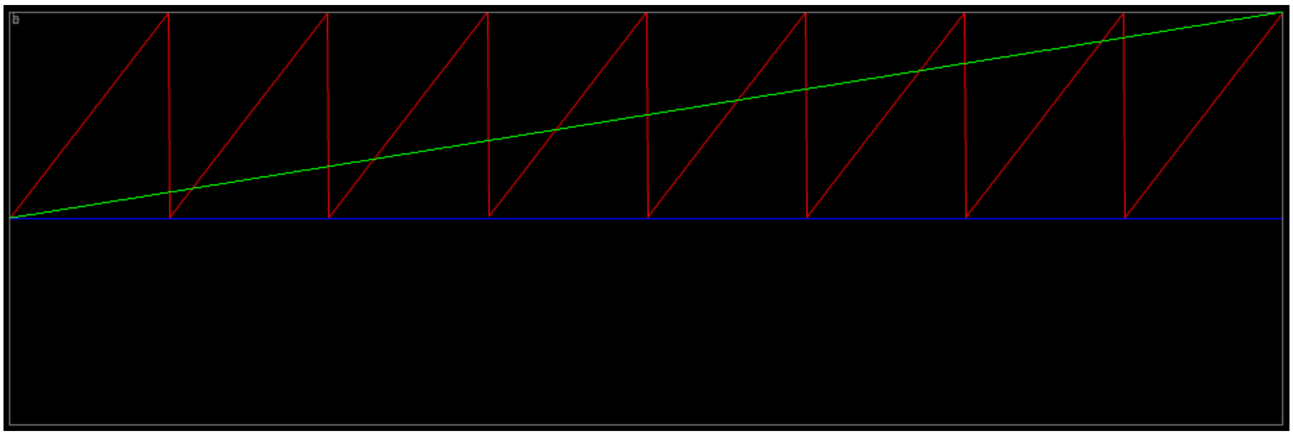
The input of our expression will be a continuous increasing ramp: $v1. You can think of it as a giant saw tooth wave signal. It grows +8000 per cycle by default and the default cycle is about one second: 1Hz / 120bpm.

The first operator you must handle is modulus: %. Modulus works as a reset. X%Y resets the X input each Y steps. To reset the saw tooth wave each cycle simply use $v1%8000.

$v1

0	%	8000	=	0
1000	%	8000	=	1000
7999	%	8000	=	7999
8000	%	8000	=	0
9000	%	8000	=	1000

...



But wait, since the output goes directly to the audio converter, it should always be somewhere between -1 and 1. So, if you divide the modulus by itself you get a 0 to 1 ramp: $v1%8000/8000.

$v1

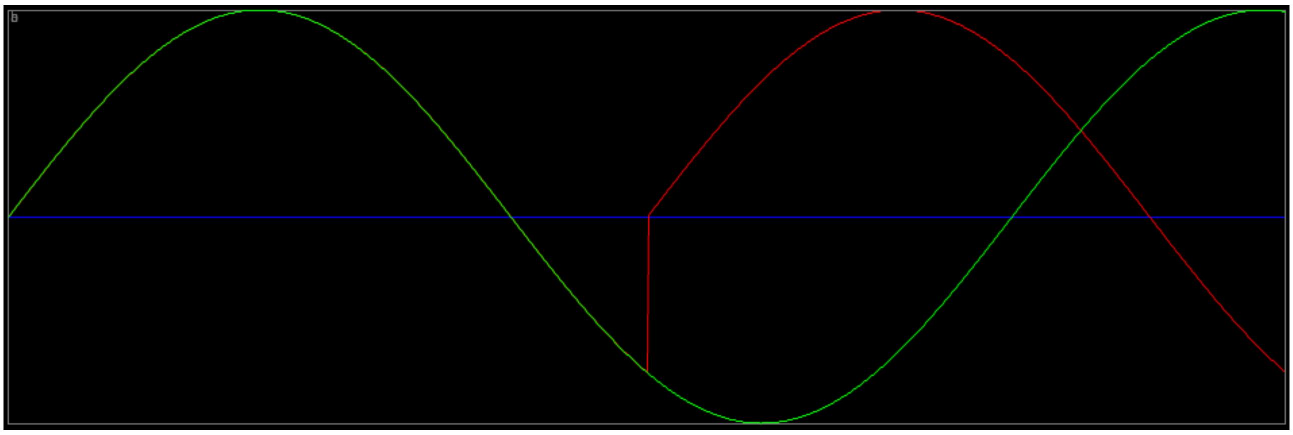
9000	%	8000	=	1000	/	8000	=	0.125
10000	%	8000	=	2000	/	8000	=	0.25
15999	%	8000	=	7999	/	8000	=	0.99

You can try smaller ramps like $v1%1000/1000 (which is a 1/8 measure) and, of course, you'll perceive it as a clear pitch if the reset frequency is on the audible spectrum, like $v1%80/80 (which is nothing but a 100Hz saw tooth wave at the default settings).

3. Sines

Sine waves are very useful in synthesis and specially on rampcoding, since the sin() and cos() functions return directly a -1 to 1 output. To get different frequencies just multiply or divide the input like in sin($v1/1000). You can also reset the input, for example, to sync an LFO as in sin($v1%4000/1000), which resets the sine every ½ cycles.

Another good advantage of working with sine is that you can play with just intonation scales, dividing and multiplying the input signal. We will see a little of this further.



4. Sequences

I figure two simple ways of making sequences which could be understood as sample and hold variations. The first is doing a simple counting. With the help of modulus and conditionals, this first way is good to construct euclidean rhythms. The second option is to evaluate a sine function at a certain beat or tempo. This second way could be rapidly adapted to generate pseudo random patterns.

4.1. Counters

The counting formula is dividing the signal in the counting measure and then applying one or more modulus to the result. Since modulus always return an integer, see what happens in $v1/1000%8:

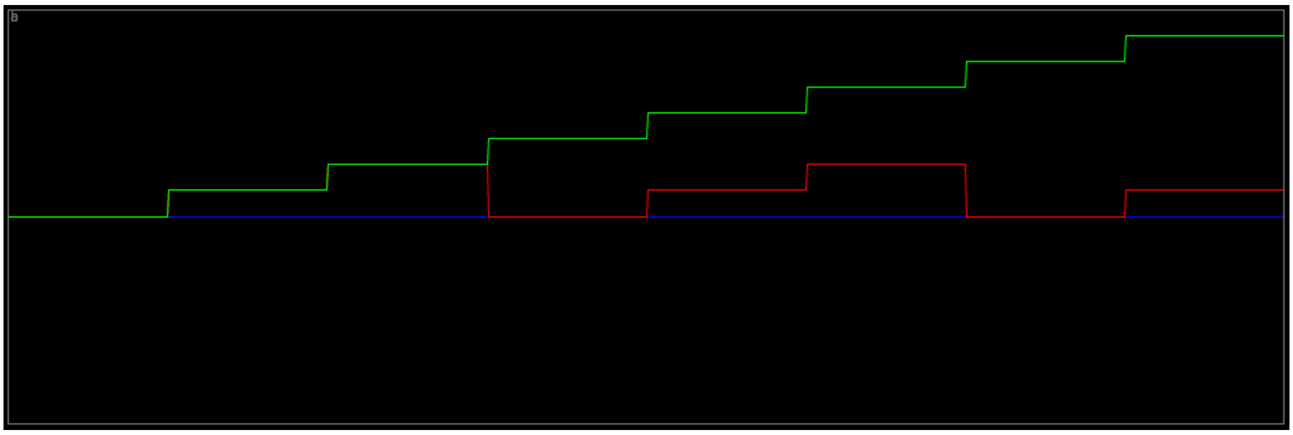
$v1

0	/	1000	=	0	%	8	=	0
900	/	1000	=	0.9	%	8	=	0
1000	/	1000	=	1	%	8	=	1

…

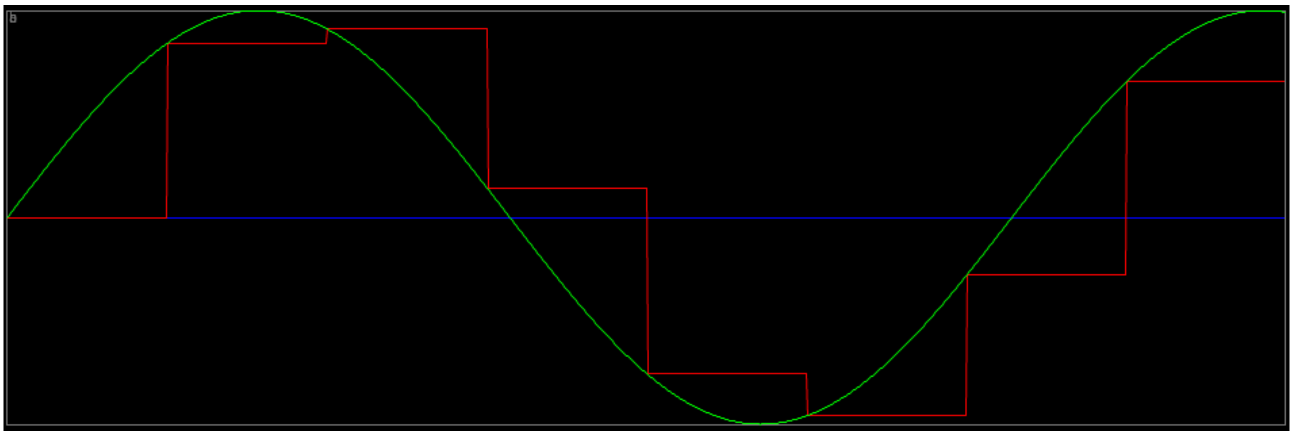
So the modulus output will increase and hold in tempo. Keep in mind that you can concatenate more than one modulus to get different patterns like in $v1/1000%8%3:

0	%	8	=	0	%	3	=	0
1	%	8	=	1	%	3	=	1
2	%	8	=	2	%	3	=	2
3	%	8	=	3	%	3	=	0
4	%	8	=	4	%	3	=	1
5	%	8	=	5	%	3	=	2
6	%	8	=	6	%	3	=	0
7	%	8	=	7	%	3	=	1
8	%	8	=	0	%	3	=	0



4.2. Pseudo randomness

My second option to generate sequences is to evaluate a sine function at a certain beat. This is the same than the previous but, instead of taking the counter result, we use it inside a sin() function like in sin($v1/1000%8). Of course, you can always make this counting more complex with multiplications, divisions, offsets, etc. Just remember that the sin() function returns a -1 to 1 output so you must scale it to whatever range that you're trying to generate.



5. Control

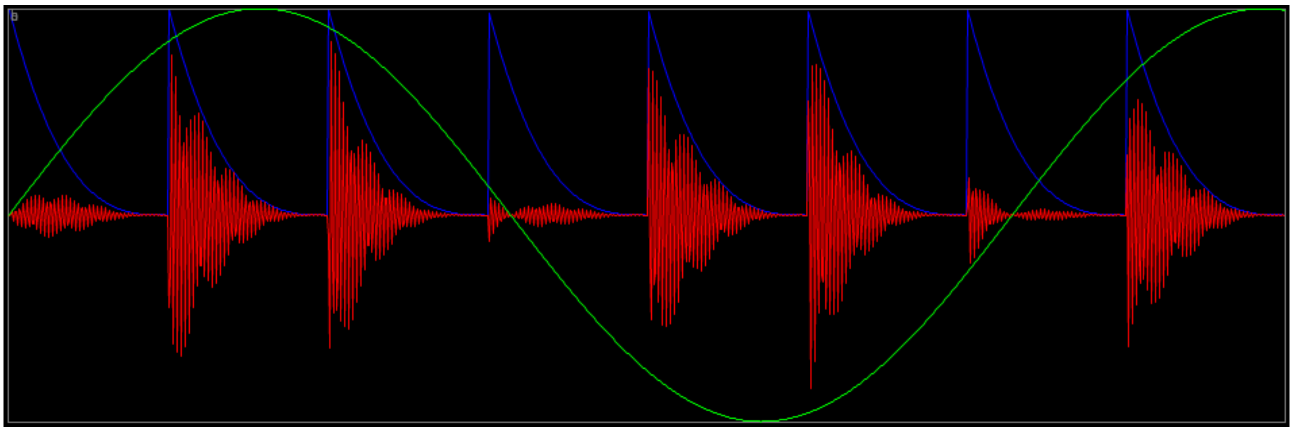
Sure that building a sound from scratch has its disadvantages in comparison to working with common instruments and presets but one of the things that make synthesis so interesting is that you can control every aspect of the sound. Let's see come basic techniques applying more or less what we just learned.

5.1. Amplitude

Remember the 1/8 subdivision of the ramp ($v1%1000/1000)? This can't be perceived as a pitch but can be used as an envelope. In fact, you can invert the ramp so you get an instant attack and then a release: 1-$v1%1000/1000.

Now we have a nice percussive envelope but we're just getting started. What if you multiply two frequencies? Yes, you're basically doing AM. Try this:

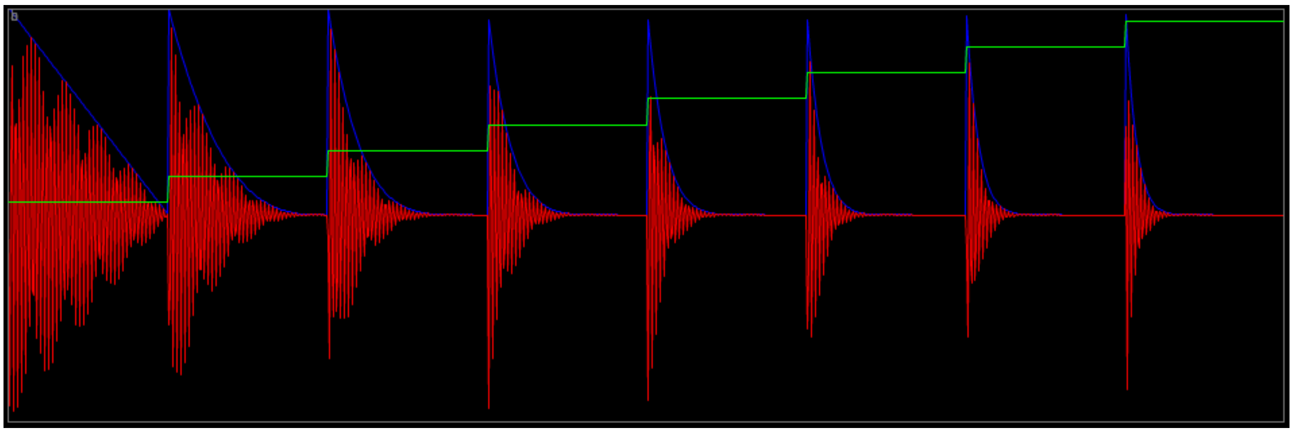
```
sin($v1)*sin($v1/1000)*(1-$v1%1000/1000)
```



5.2. Release

If you power the envelope output you can sharp the release: pow($v1%1000/1000,3). But let’s put a sequence on that envelope sharpness. Start with a simple counter:

sin($v1)*pow(1-$v1%1000/1000, $v1/1000%8*2+1)



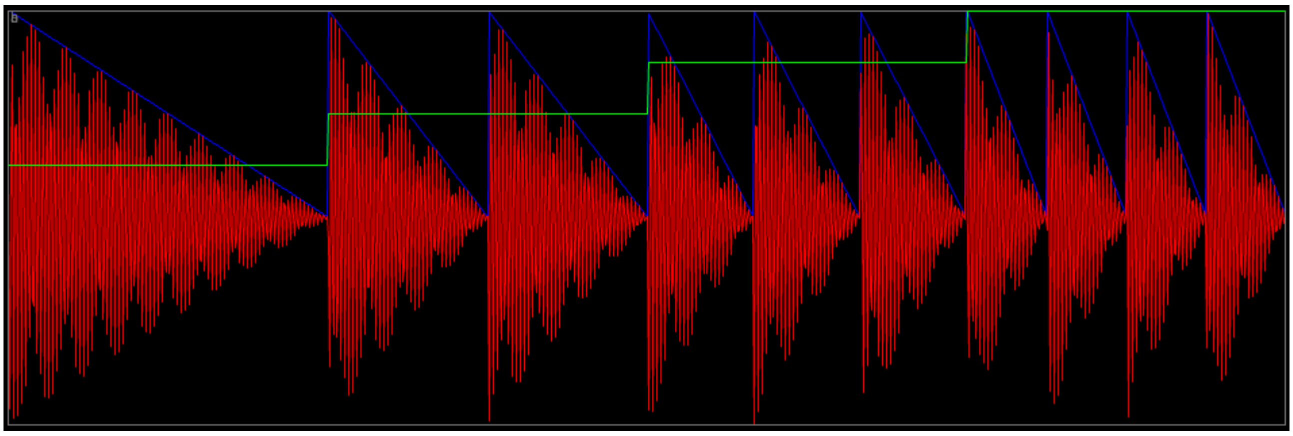
Here we put a *2 so the change is more notorious and a +1 to avoid the 0 release step.

Note: If you’re sending messages to Pure Data, remember that “,” should be preceded with an escape sign like this “\,”.

5.3. Measure

We’re working with expressions and they’re totally recursive. So, instead of writing $v1%1000, you could also write $v1%(8000/8) and get the same result. This sounds silly but it is very useful for rare measures like septuplets: $v1%(8000/7). But, one step further, you can put some counting or whatever on that division and make a changing measure sequence. Try this:

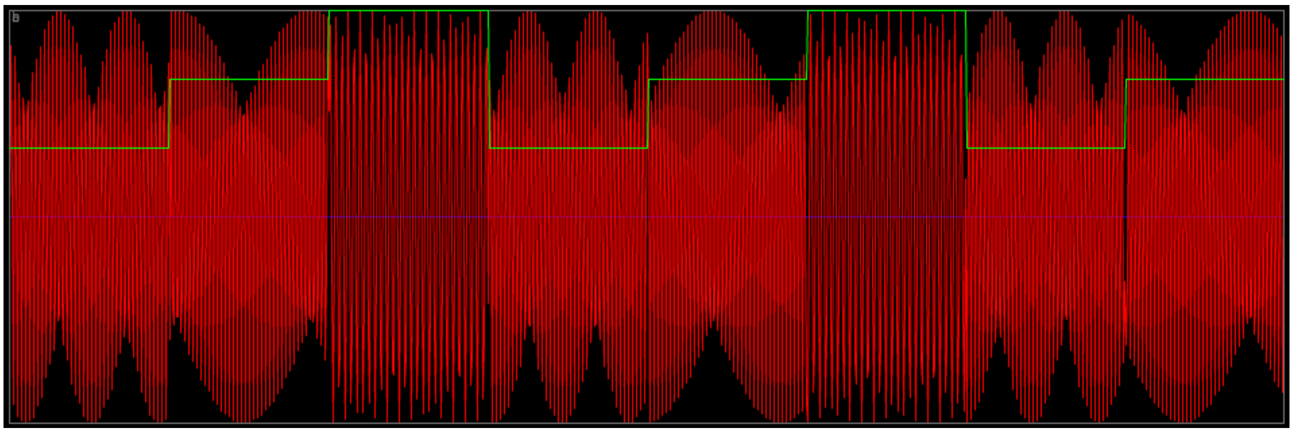
```
sin($v1)*(1-$v1%(2000/(1+$v1/2000%4))/(2000/(1+$v1/2000%4)))
```



5.4. Scales

As we already mentioned, dividing and multiplying the incoming signal inside a sin() function is an easy way to handle notes. If you don't know just intonation, it worth to take a look. Do some experiments with that knowing that there is an harmonic relation between frequencies that are multiplied and/or divided by integer numbers. Try this basic arpeggiator and start experimenting with its parameters:

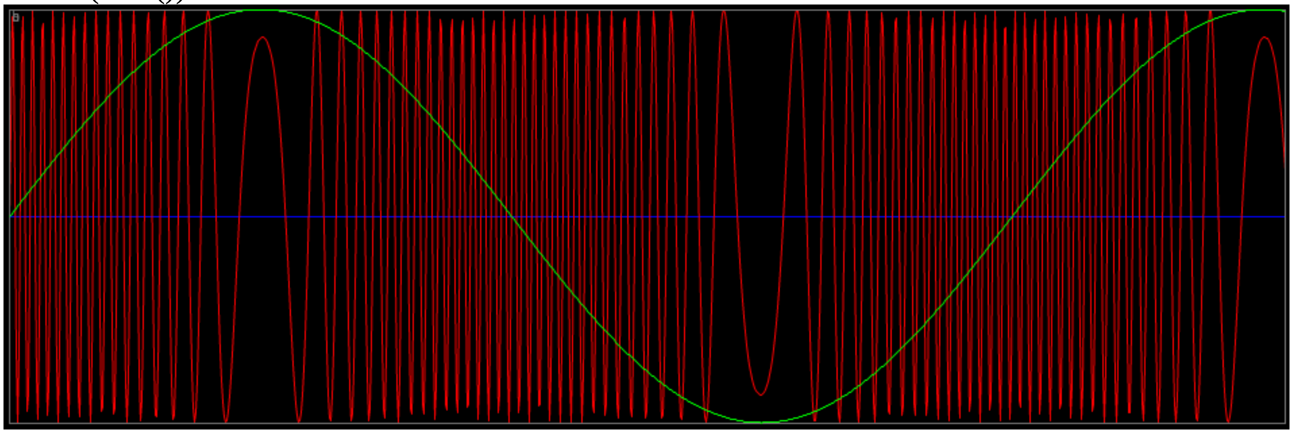
```
sin($v1/($v1/1000%3+1)/2)
```



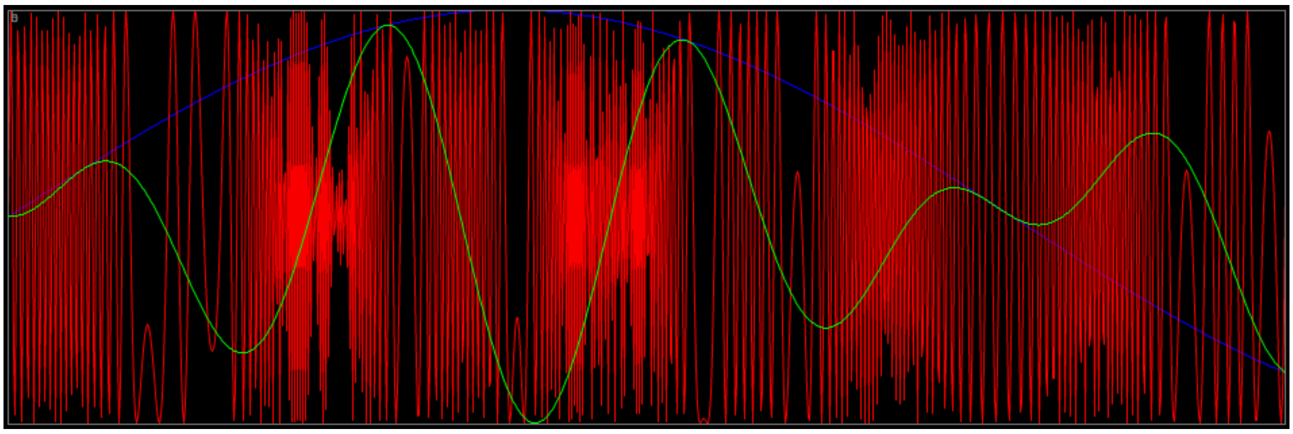
Did you notice the counter inside the sin() function?

5.5. Frequency

A super powerful synthesis technique is FM and it is very easy to build it in expressions following the sin(+sin()) formula.



This is a two operator FM algorithm: sin($v1/1000+sin($v1/1000)*100). The *100 just increases the modulator amplitude so the effect is exaggerated. This has almost no limit, you can put more and more operators anywhere. Like here: sin($v1/80+sin($v1/80)*sin($v1/6400)*10). A third operator was add to control the modulation amplitude. Nice. These two examples have a 1:1 ratio. Try different ratios and audible base frequencies like sin($v1/8) and so. Another good idea is to apply a sharp envelope to the third operator to control the amplitude of the modulation like sin($v1/80+sin($v1/80)*sin($v1/6400)*10*pow(1-$v1%1000/1000,10)).

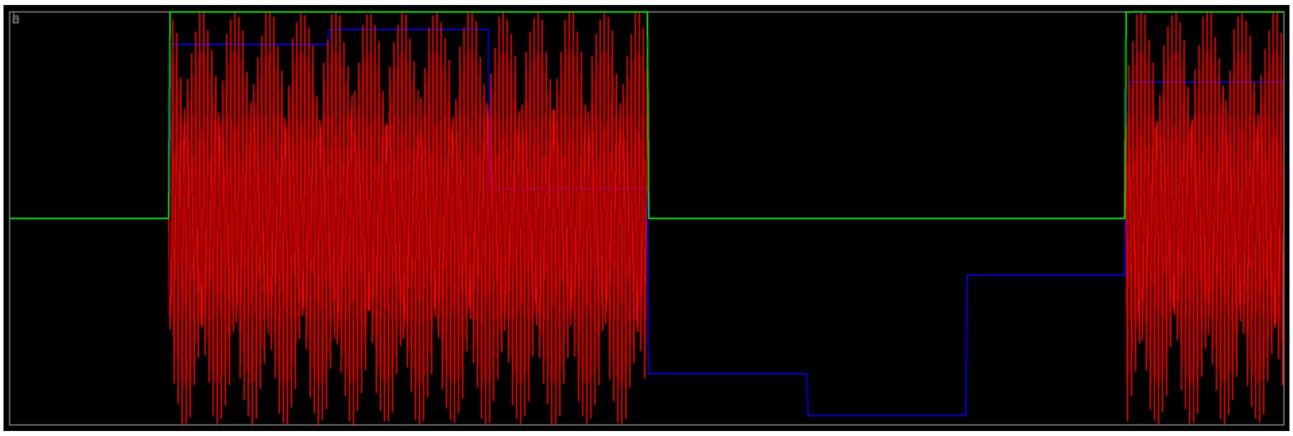


Tip: High modulation amplitudes are noise, a quick shortcut to noise could be sin($v1*sin($v1)).

5.6. Trigger

Ok, time to say something about conditionals. Expressions can handle boolean operators (<, >, ==, !=, &, |, etc). They are very useful to control triggering. See this example:

sin($v1)*(sin($v1/1000%8)>0)

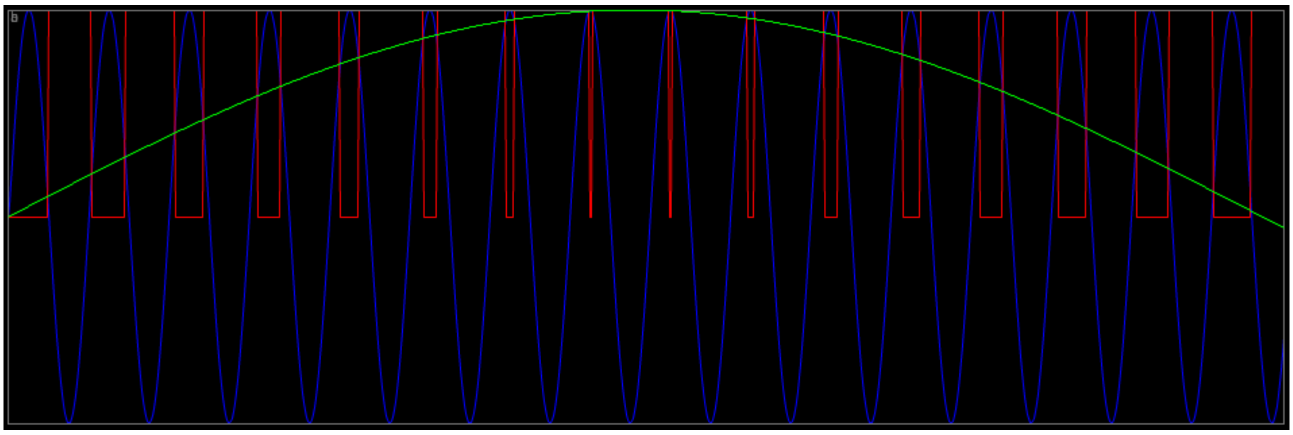


The amplitude of the sine will be equal to the output of the condition (true=1 or false=0). Of course, this is just a clear example, usually you’ll probably want to add an envelope like sin($v1)*(sin($v1/1000%8)>0)*(1-$v1%1000/1000). A more predictable way of handle triggering is with counters like sin($v1)*($v1/1000%8==0). This will trigger every first 1/8 measure of the cycle.

5.7. Pulse

Conditionals and operators utility goes beyond triggering. I'll give you one more example. The first thing I can think of is square waves. A sin() with a conditional is a good way to create a square wave like (sin($v1/8)<0)). It allows you to control the pitch and the pulse width! See these PWM examples:

```
sin($v1/8)<sin($v1/2500)
sin($v1/80)<sin($v1/2500)
```

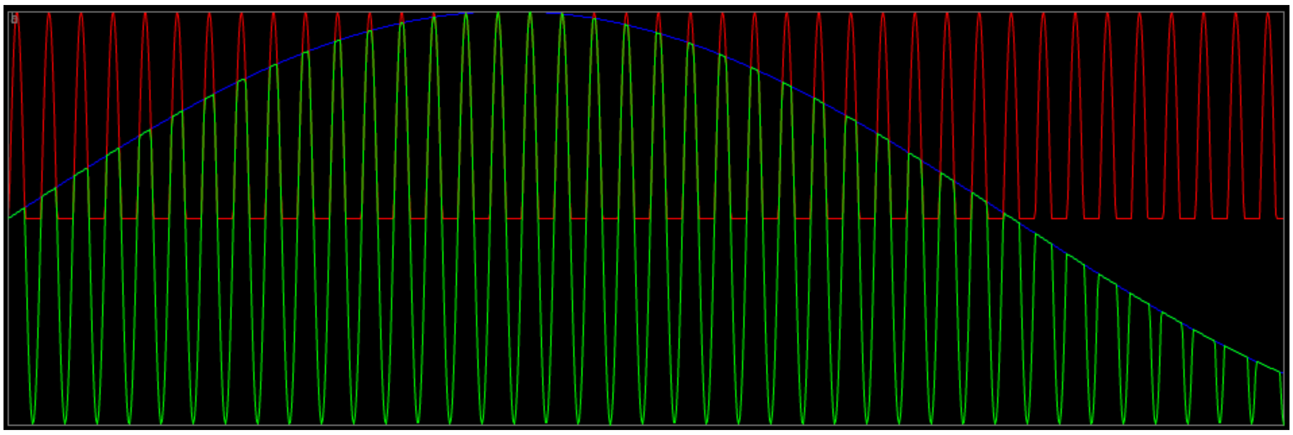


6. Distortion

There are lots of ways to distort a signal with expressions, I will show you two. The simplest is by digital clip, this is clipping the signal at a certain amplitude, a very old trick that sounds ok and let you modulate the distortion. The second one is a degrade with a kind of a bit crusher result. This second one is a little more complicated but has a crazier behavior.

6.1. Clip

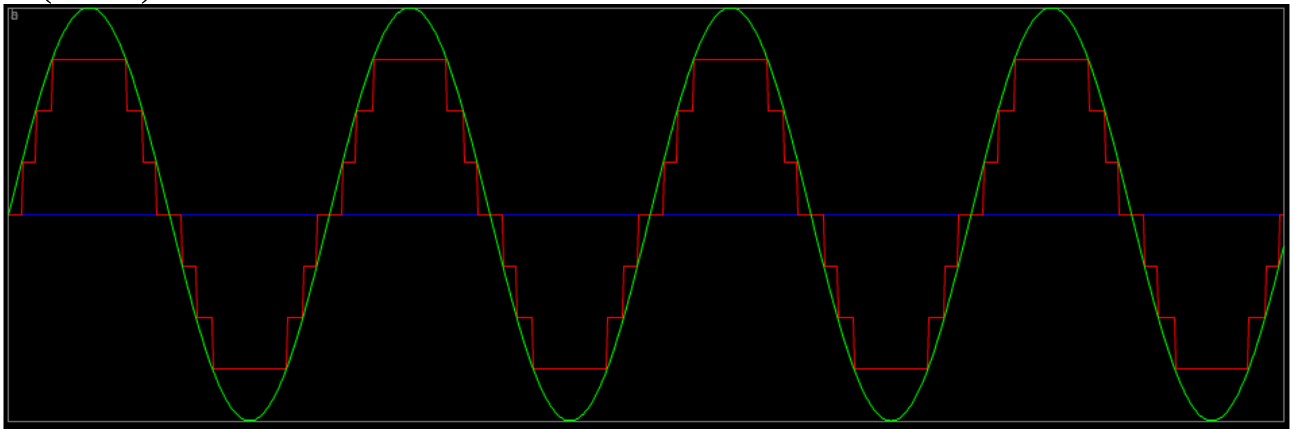
You can clip the signal at a minimum or a maximum point. max(sin($v1/32),0) will return all the values higher than 0, cutting the half of the signal. You can try different clipping points but, also, control this point with modulations or envelopes like min(sin($v1/32),sin($v1/2000)) or min($v1/32,($v1%4000/4000)) and things like that.



6.2. Degrade

As we learned, modulus returns an integer value so it is kind of a sample and hold or pixelate effect already. All you have to do if you want to degrade a signal is making it big enough so then you can use modulus to loose definition. See, the sin() function, for example, has a -1 to 1 output. If you multiply it by 10 you get -10 to 10. But the values that the function returns are not integers, there will be many values between -9 and -8 like -8.99, -8.5, -8.333, etc. So, if you apply a modulus you'll be like messing with its resolution. Of course, you'll have to divide after so everything is again mapped between -1 and 1. Like here:

sin($v1/32)*4%4/4



And a final comment before you leave this guide and start experimenting alone or checking other examples. The stranger things happen when you reduce drastically the amplitude multiplication with the modulus. Try this: sin($v1/32)*100%10/10