joachim heintz
workshop at TIEMF 2021
24 february (I), 3 march (II), 10 march (III)

Views on Sine Waves

I. A Sine Cloud

Goals of this workshop:

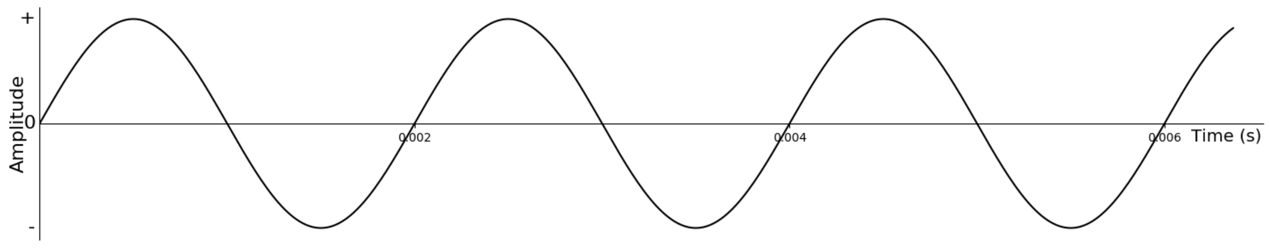
- Explore different views to sine waves.
- Develop a model (here: a percussive sine tone, and a cloud of percussive sines) in different directions.
- Discuss some elements of electronic music which are important not only for sine waves: linear versus exponential envelopes, random choices and deviations, ways to work with volume and pitch, usage of arrays for partials and for musical structures.
- Experience how musical ideas can be transformed into code.

For the first part of this workshop, we will start with a simple sine, and end at a cloud of percussive sines which can be shaped in different ways.

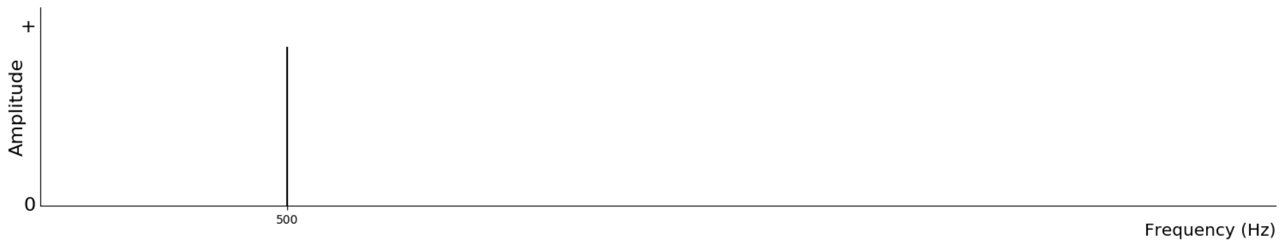
All examples for this part can be downloaded as .zip file here.

1 Three ways to look at a sine

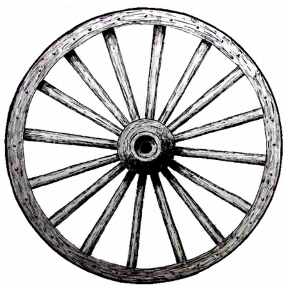
The most common way to look at a sine in electronic music is an amplitude versus time graph.



When we look at a sine in the frequency domain, we see nothing but a thin line.

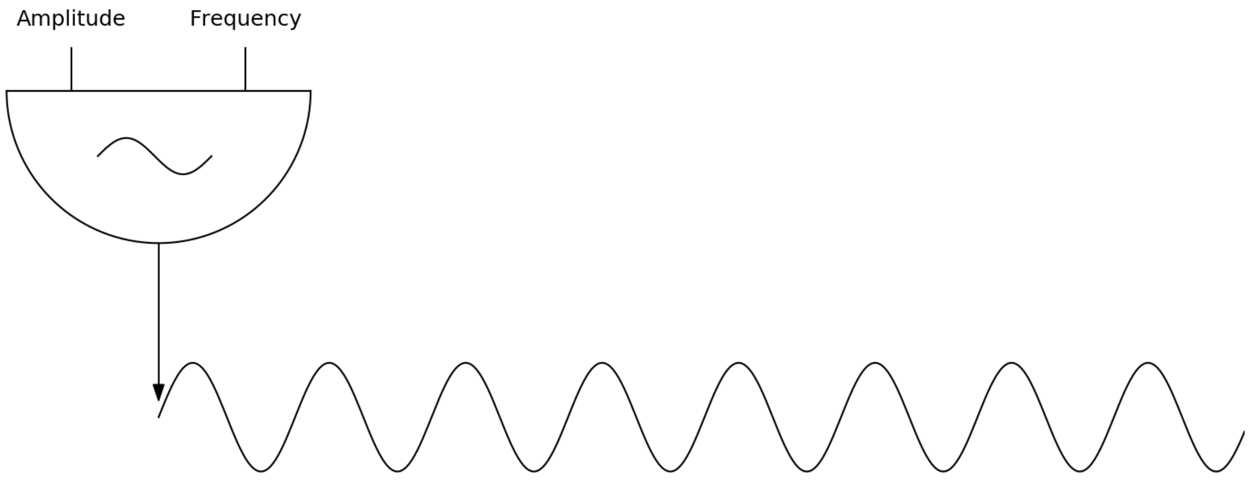


But this is also a way to look at a sine:



All three ways are valid aspects of a sine.

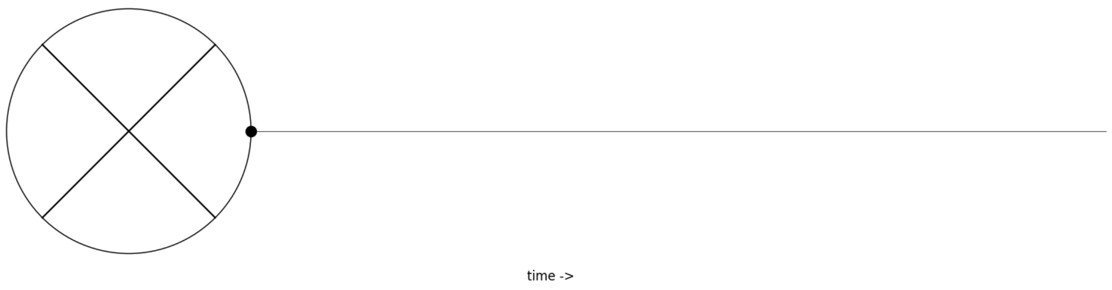
The first way is the usual way we look at a sine, and we produce a sine in electronic music by an oscillator, determining amplitude and frequency.

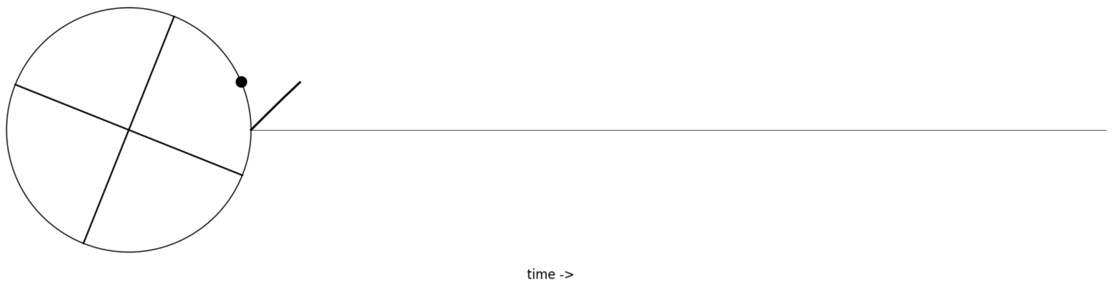


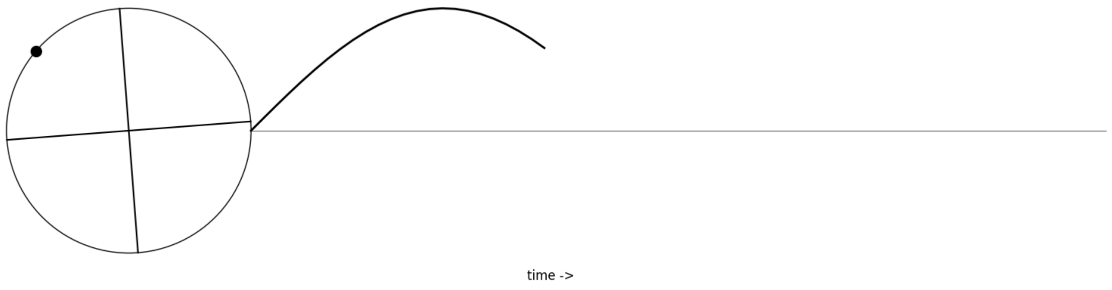
The second way we will explore in the second part of this workshop.

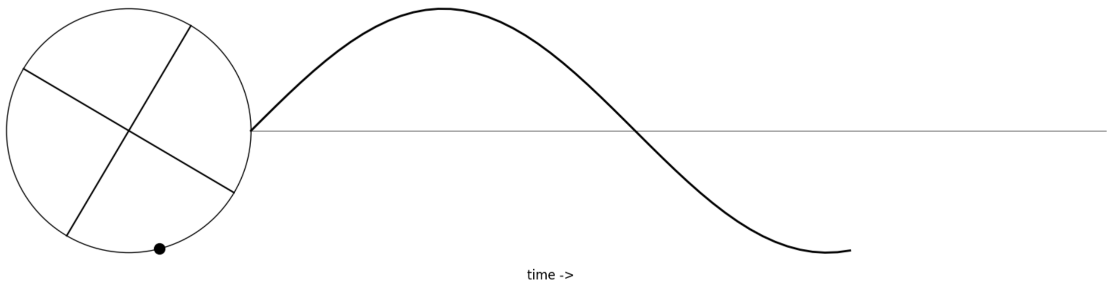
The third way is actually the classical way to create a sine, as movement of a point on a circle. Imagine we watch a point on a wheel while the wheel is moving,[1] and plot the point's movement over time. Here are five snapshots:

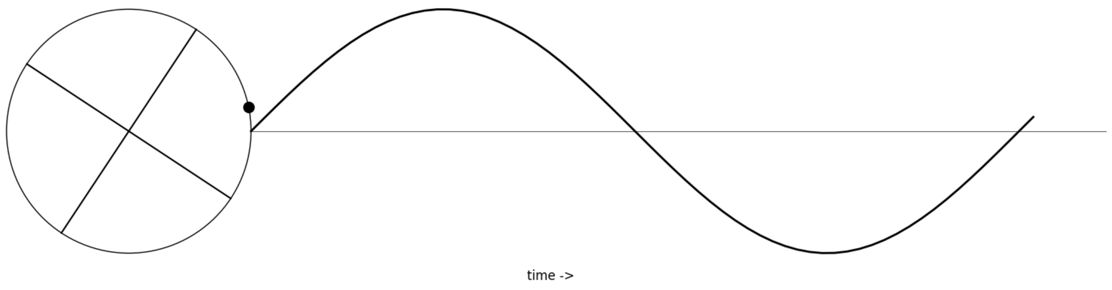
1 here counterclockwise (following the convention in mathematics)





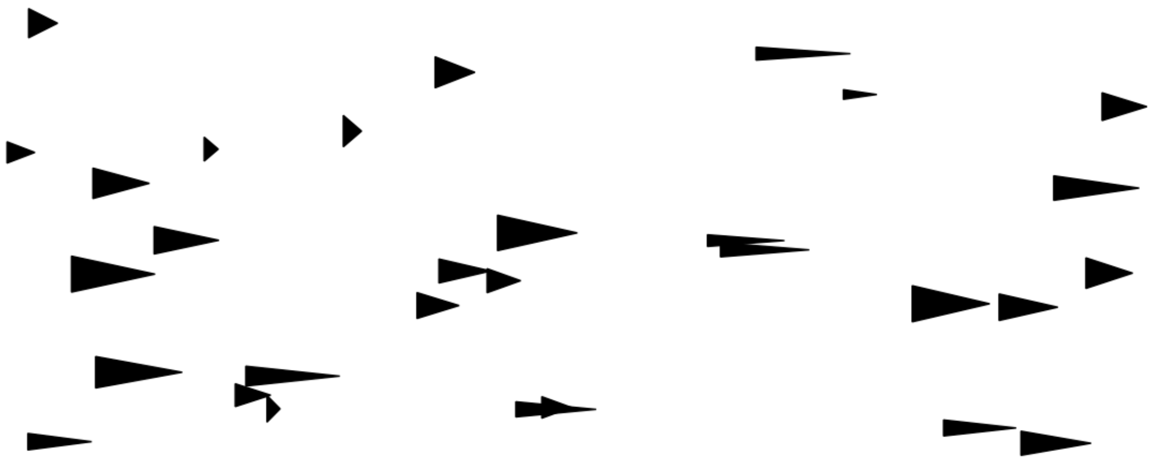






But moreover, a wheel is a *model*. A model can give us new ideas. A wheel can move slower or faster, it will usually show some irregularities, and so on.

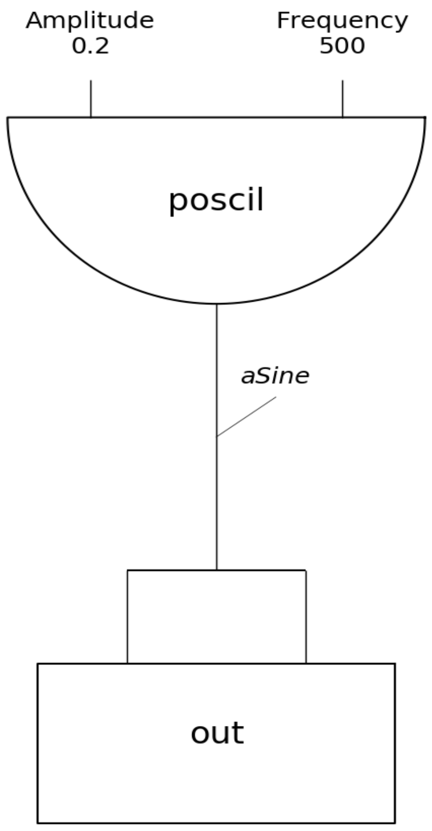
The whole workshop is built on the idea of a model — not the model of a wheel, but the model of a percussive tone, consisting of one or more sines, and also the model of a cloud, consisting of percussive sines in different sizes, durations, constellations. For instance:



Models, or objects, or *Gestalten*, can also be seen in a musical way, as rhythmic patterns, harmonic structures or gestural ideas which can be modified, for instance by extending or compressing, by permuting or repeating. We will explore this in the third part of this workshop.

2 Simple Sine

For now we will start with the first view on the sine, as amplitude versus time. This is the flow chart for a Csound instrument to create it:



And this is the Csound code for it: -> I_01.csd

```
instr SimpleSine
 aSine poscil 0.2, 500
 out aSine, aSine
endin
```

3 Alternative Code Style: Functional

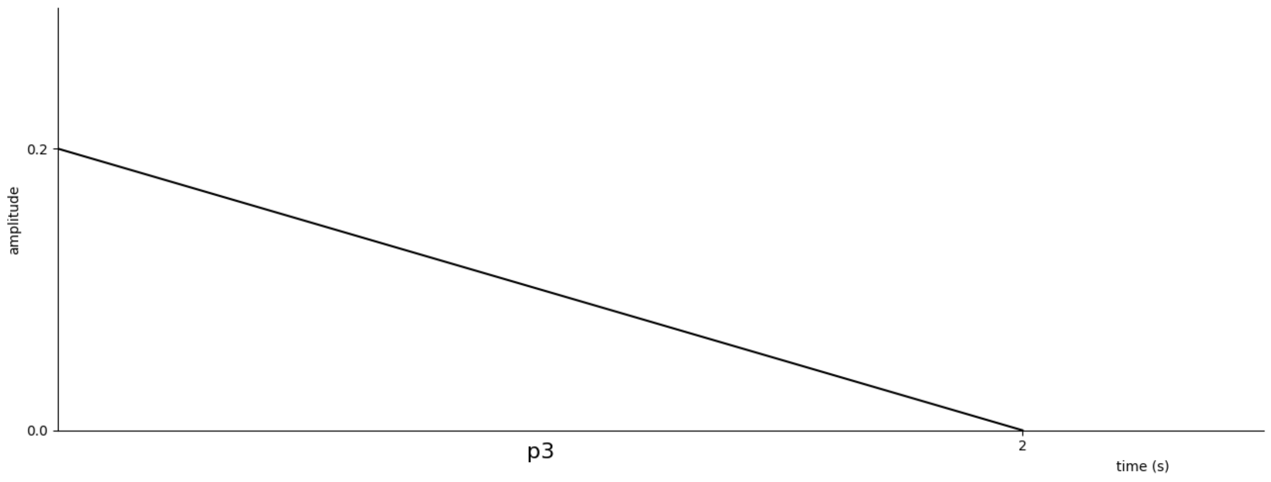
If you are used to Python or a similar programming language, you can also write Csound code in this style. The above code would then look like this: -> I_01_functional_style.csd

```
instr SimpleSine
 aSine = poscil:a(0.2,500)
 out(aSine,aSine)
endin
```

I will continue in classical Csound style here, and from time to time introduce some code snippets in functional style.

4 Apply Linear Envelope

For a percussive tone, we need an envelope for the amplitude. Here a continuous decay from the maximum amplitude value (0.2 in the previous example) to zero.

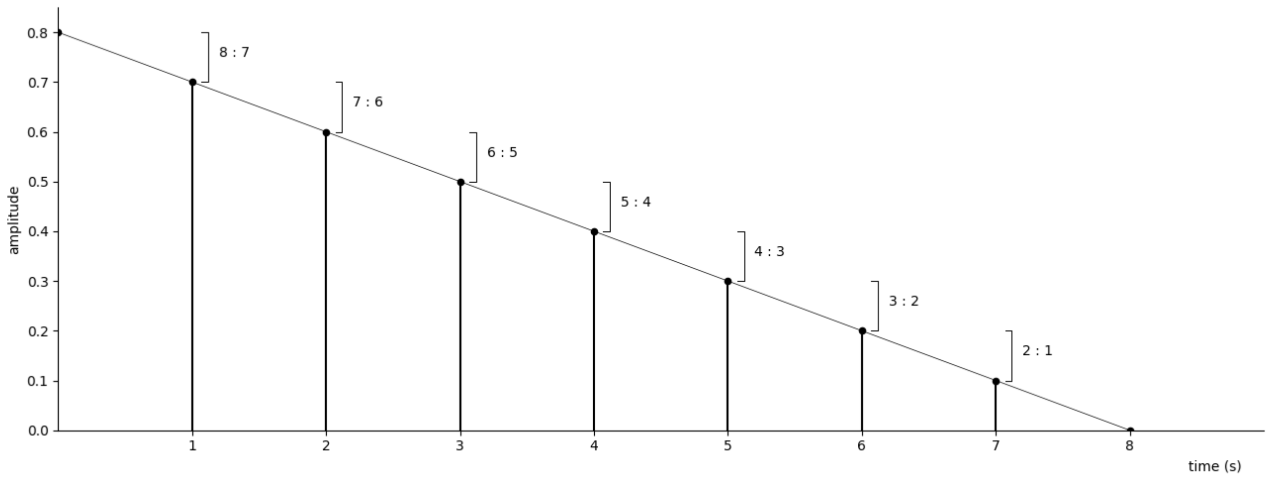


This is the Csound code: -> I_02.csd

```
instr LinearEnv
 aEnv linseg 0.2, p3, 0
 aSine poscil aEnv, 500
 out aSine, aSine
endin
```

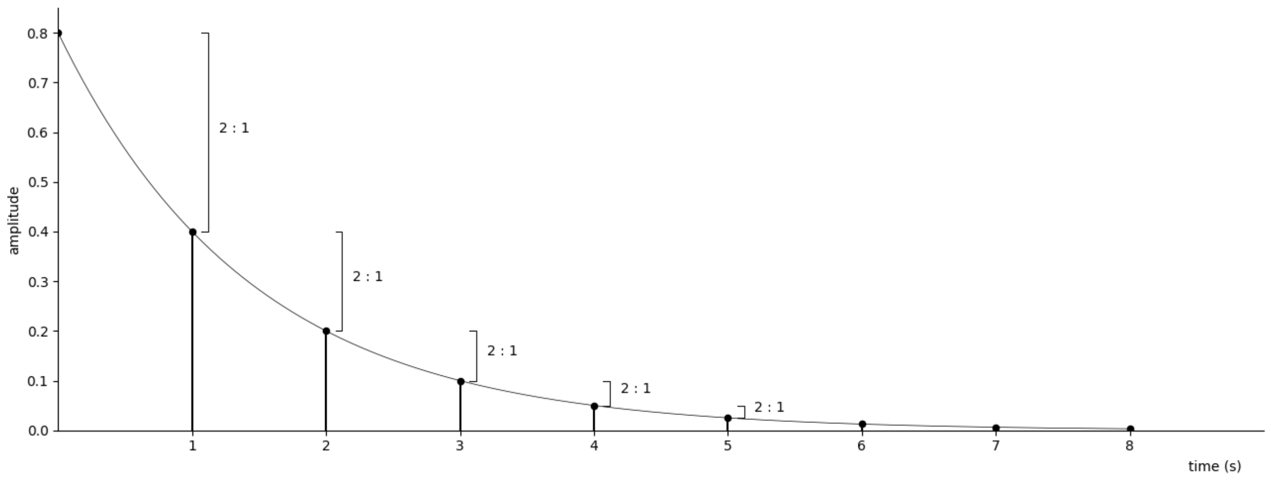
The problem here is that what *looks* like a smooth and continuous diminuendo or fade-out, does not *sound* like it. The reason is actually quite interesting and belongs to the way we perceive the physical reality by our senses. In short: We perceice *proportional*, as ratios.[2] Here: We perceive something as *always in the same amount softer*, if the amplitude has the same proportion from this step to the next, as it had from this step to the previous.

The line we created above does not show the same poportions from one step to the next. Imagine a linear decay from ampltude 0.8 to 0 in 8 seconds:



We do not at all detect the same proportions from one step to another, but quite different ones. Actually the proportions become bigger and bigger. This is the reason why we hear the decay as small at the beginning, and big at the end. We perceive the decay as "too slow" at the beginning and as "too fast" at the end.

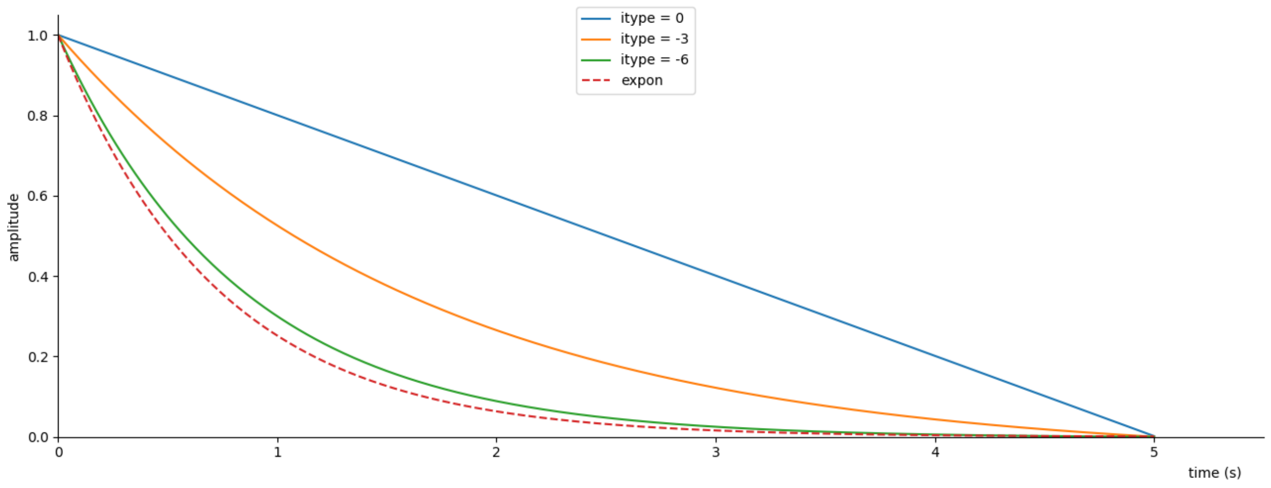
If we want to get the same proportions, in the most simple case the proportion 2:1, for each step, the graph will look like this:



2 This is known as Weber-Fechner-Law. In music, it applies for both, intensities and pitches.

5 Apply Exponential Envelope

This is what is called *exponential*, and can be exponential rise, or as in our case, exponential decay. The traditional way to use it in Csound is the *expon* opcode. I personally prefer the *transeg* opcode which offers a simple way to shape the curve. This is done with a parameter which is called *itype* in the Csound Manual. Here come different shapes for a decay from 1 to 0 in five seconds, depending on different values for *itype*:



Usually i choose something between -3 and -6 for *itype*. Make your own choice according to what you prefer.

This is the new code now: [-> I_03.csd](I_03.csd)

```
instr ExpEnv
 aEnv transeg 0.2, p3, -4, 0
 aSine poscil aEnv, 500
 out aSine, aSine
endin
```

6 Apply Different Volumes

We define a variable to apply different volume levels. As result of the way we perceive intensity, we should not use raw amplitude input, but use the *Decibel (dB)* scale. If the maximum amplitude is set to 1, this is 0 dB.[3] Roughly, each -6 dB results in half the amplitude.[4] This is an overview about the usual range of hearing, from maximum volume (0 dB) to very soft (-66 dB):

3 It is exactly this what the header statement `0dbfs = 1` means.
4 To be precise, -6 dB is not 0.5 but 0.501187. See https://flossmanual.csound.com/basics/intensities.

dB	amp	dB	amp
0	1.000000	-36	0.015849
-6	0.501187	-42	0.007943
-12	0.251189	-48	0.003981
-18	0.125893	-54	0.001995
-24	0.063096	-60	0.001000
-30	0.031623	-66	0.000501

As we want to insert dB as users, but the oscillator requires amplitude input, we must *convert* the dB input to amplitudes. This is done by the *ampdb* opcode in Csound.[5]

This is the code now: -> I_04.csd

```
instr ExpEnv
 ;get dB input
 iDb = -14
 ;convert dB to amplitude
 iAmp ampdb iDb
 aEnv transeg iAmp, p3, -4, 0
 aSine poscil aEnv, 500
 out aSine, aSine
endin
```

7 Apply Different Pitches

Similar to volume, we want an easy and musical was to înput different pitches. We will not use the Hertz frequency input directly, by two reasons. Firstly, also in pitch recognition we perceive in proportions. We call the proportion 2:1 an octave, and we perceive each octave as equal intervall. But the frequency distances are not at all equal. Look at this overview for the note A:

Note name	A1	A2	A3	A4	A5	A6	A7	A8
Frequency (Hz)	55	110	220	440	880	1760	3520	7040

The proportions between each A and its neighbour are always 2:1, but the frequency distance is very different: 220 Hz from A3 to A4, but 440 Hz from A4 to A5. In other words: It is quite easily misleading in musical terms to work with frequencies.[6]

And secondly, we are used to think in traditional pitches and notation. Who knows which pitch approximates to the 500 Hz we used so far? A well established way to work with pitches is the

5 Usually used as converter in functional style. So ampdb(-6) is exactly the same as 0.501187.

6 This is shown for instance when we apply a glissando from one frequency to another frequency in using a linear transition, for instance kGlissando linseg 880, 3, 440. This will behave quite similar to the linear envelope we tried in I_02.csd: move "too slow" at first, and "too fast" at the end. For more explanations and examples, see https://flossmanual.csound.com/basics/pitch-and-frequency

MIDI not system. Here, the middle C (C4 in American system, or c' in German) gets the number 60. Each semitone has a distance of 1 to the next one. An octave is always 12 as difference, and the overview above with MIDI notes rather than frequencies is as follows:

Note name	A1	A2	A3	A4	A5	A6	A7	A8
MIDI note number	33	45	57	69	81	93	105	117

In recent Csound, we convert MIDI note input to frequency by the *mtof* opcode. (If you have an older Csound version, use *cpsmidinn* instead.)

This is the code which adds pitch input as MIDI: -> I_05.csd

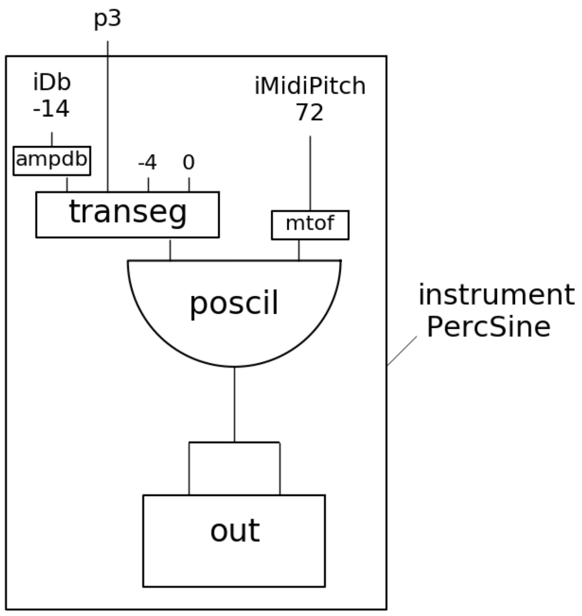
```
instr ExpEnv
 iDb = -14
 ;get midi key input for pitch
 iMidiPitch = 72
 iAmp ampdb iDb
 ;convert MIDI note to frequency
 iFreq mtof iMidiPitch
 aEnv transeg iAmp, p3, -4, 0
 aSine poscil aEnv, iFreq
 out aSine, aSine
endin
```

We can now compact the code by using functional style for the conversion from dB to amplitudes, and from MIDI to frequency: -> I_06.csd

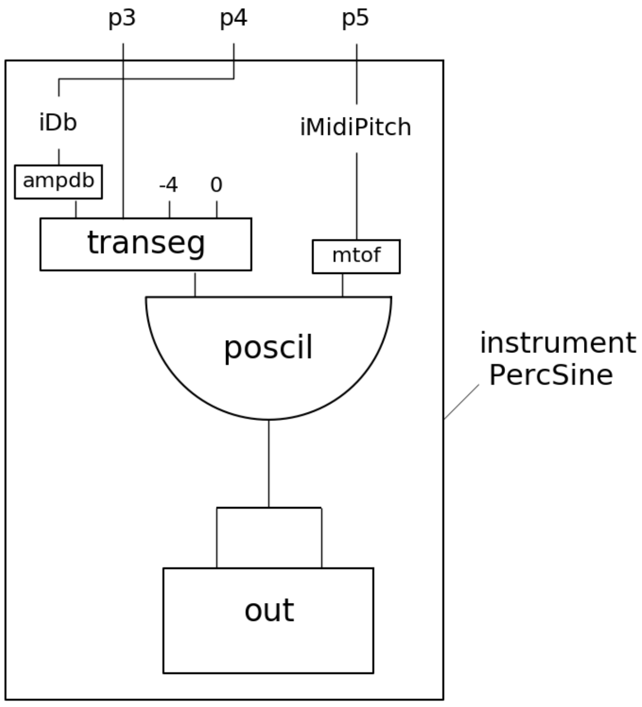
```
instr PercSine
 iDb = -14
 iMidiPitch = 72
 aEnv transeg ampdb(iDb), p3, -4, 0
 aSine poscil aEnv, mtof(iMidiPitch)
 out aSine, aSine
endin
```

8 Move the Volume and Pitch Input to Outside the Instrument

So far all modifications of volume and pitch we only do inside the instrument.



But if we want to use this model in a flexible way and connect it with other units, we must shift the major modifications to outside the instrument.



In Max or PD we would call this *inlets*. If we define a function in a programming language, we call it *arguments*. For an instrument call in Csound, we speak of *parameter fields* (or abbreviated *p-fields*). The first three parameters in an instrument call (or instrument event) are always fixed as

1. instrument number or name (p1)
2. start time (p2)
3. duration (p3)

In other words: We can add more p-fields with arbitrary meaning. In our case we choose to pass the volume as fourth parameter (p4) and the pitch as fifth parameter (p5). So now our instrument looks like this: -> I_07.csd

```
instr PercSine
 ;get dB input for volume as parameter 4 in the instrument call
 iDb = p4
 ;get midi key input for pitch as parameter 5 in the instrument call
 iMidiPitch = p5
 aEnv transeg ampdb(iDb), p3, -4, 0
 aSine poscil aEnv, mtof(iMidiPitch)
 out aSine, aSine
endin
```

9 Call the Instrument from Another Instrument

The traditional way to continue now would be to add score lines in which we call the instrument at different start times (p2), for different durations (p3), with different volumes (p4) and different pitches (p5). We could either write these score lines by hand, or we could use another application to produces these lines.[7]

We will go another way here. We will send instrument calls from another instrument. Let us start with a simple example: -> I_08.csd

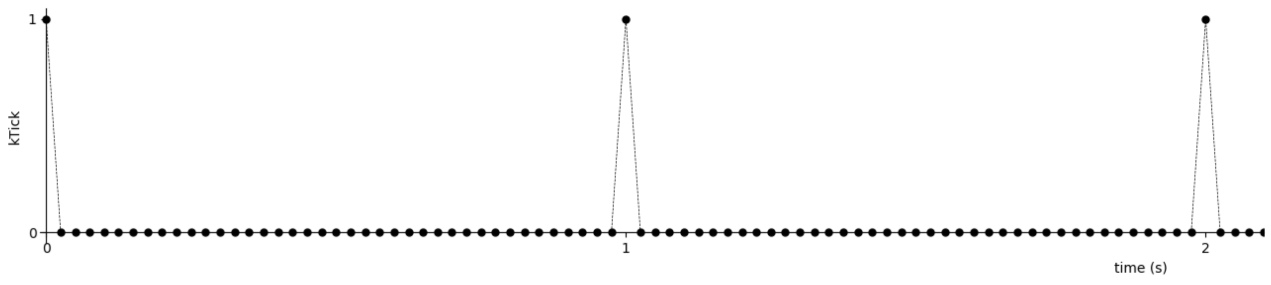
```
instr Cloud
 ;create a tick once per second
 kTick metro 1
 ;if it ticks ...
 if kTick == 1 then
  ;... call instrument "PercSine" with the given parameters
  schedulek "PercSine", 0, 1, -10, 60
 endif
endin
```

The metro opcode acts like a metronome: It outputs "ticks" at a certain frequency. Each tick is 1, and if there is no tick, the output is 0. So for the code line

```
kTick metro 1
```

the kTick signal looks like this:

7 There were and are numerous ways and applications for it, see also https://flossmanual.csound.com/miscellanea/methods-of-writing-csound-scores.



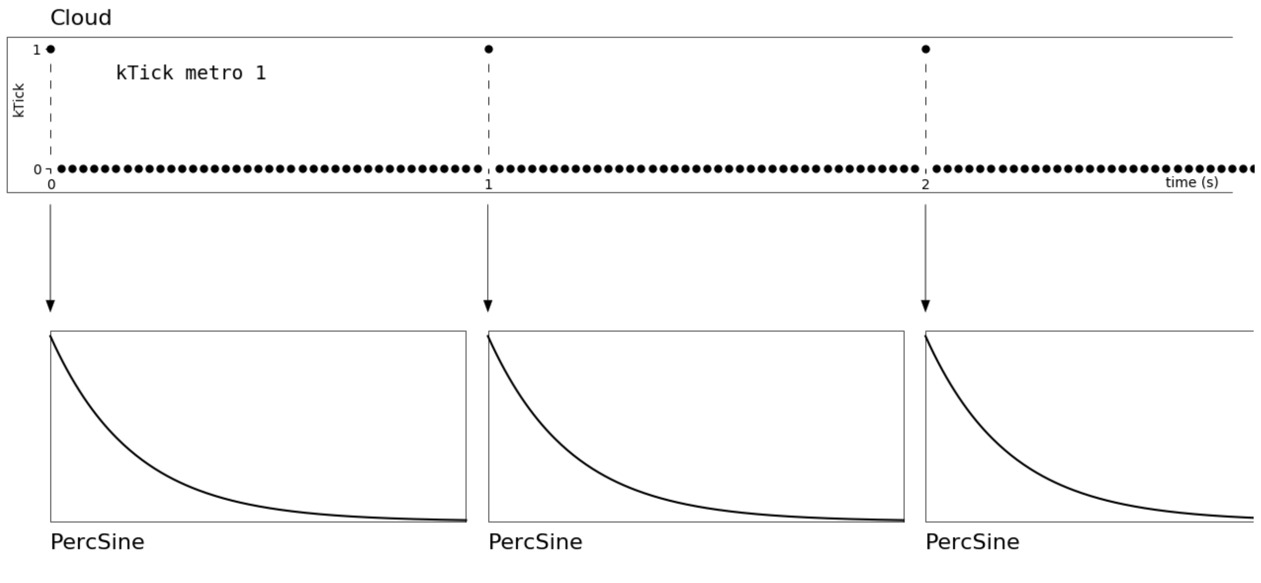
Now we can call an instrument for each tick. The sentence ...

> *If there is a tick, then call instrument "PercSine" with these parameters:*
>
> > *start time (sec) = 0 (p2) (0 means immediately, without delay)*
> > *duration (sec) = 1 (p3)*
> > *volume (dB) = -10 (p4)*
> > *pitch (MIDI) = 60 (p5)*

... looks like this in Csound code:

```
if kTick == 1 then
 schedulek "PercSine", 0, 1, -10, 60
endif
```

And this happens (note that we only have one instance of the *Cloud* instrument, but multiple instances of the *PercSine* instrument):



Note: The *schedulek* opcode was introduced in Csound 6.14. If you use an older version of Csound you must use the *event* opcode instead. The call to the instrument would then be ...

```
event "i", "PercSine", 0, 1, -10, 60
```

... rather than:

```
schedulek "PercSine", 0, 1, -10, 60
```

Exercise: Modify the metro frequency and listen to the result. Also modify the other parameters, and listen to the result.

What we did here is simple, but it has the potential to generate structures in many ways, without writing a single line of score. We will at first develop it to a cloud-like structure. In the third part of this workshop we will discuss other possibilities.

10 Random Pitches in a Certain Range

As first step we will generate a different pitch for each instrument call. We set a minimum and a maximum possible pitch, and between these boundaries we let the *random* opcode choose. We will make this random choice exactly in the moment when our if-condition is true, or in other words: when we call the "PercSine" instrument. So this part of the code will now look like this:

```
 if kNextEvent == 1 then
  kPitch random 50, 70
  schedulek "PercSine", 0, 1, -10, kPitch
 endif
```

Note: It is a common mistake to write

```
     iPitch random 50, 70
```

instead of

```
     kPitch random 50, 70
```

This would generate only *one* random value for the whole duration of the instrument.[8]

This is the code now in total: -> I_09.csd

```
instr Cloud
 kNextEvent metro 1
 if kNextEvent == 1 then
  ;generate random pitch between midi note 50 and 70 ...
  kPitch random 50, 70
  ;... and send it to the called instrument
  schedulek "PercSine", 0, 1, -10, kPitch
 endif
endin
```

Exercise: Change the numbers for minimum and maximum pitch and listen to the different results.

11 Set Random Seed

When you run the code, stop after some seconds, and run again, you will notice that the random pitches are exactly the same as in the previous run. This is not a Csound bug, but it is intended. Csound uses the *seed* opcode to set a global seed. In the manual page we read:

8 See https://flossmanual.csound.com/csound-language/initialization-and-performance-pass for an extended explanation about the different between *i* (initialization) and *k* (control) variables.

"When specifying a seed value, ival should be an integer between 0 and 2^{32}. If ival = 0, the value of ival will be derived from the system clock."[9]

In other words: Only if we set *seed 0* we will always get another random result than in the previous run. I recommend to always set *seed 0* in the orchestra header if you are working with random opcodes. If you render a piece and want to reproduce a certain version, you can change it to *seed 1* or *seed 2* or any other number.

12 Random Volumes in a Certain Range

To get different random volumes for each instance of the "PercSine" instrument which we call is quite similar to what we did for getting random pitches. -> I_10.csd

```
if kNextEvent == 1 then
  kPitch random 50, 70
  ;generate random volume between -30 and -5 dB ...
  kVolume random -30, -5
  ;... and send it to the instrument which we call
  schedulek "PercSine", 0, 1, kVolume, kPitch
 endif
```

Exercise: Compare how different the musical structure becomes if you choose a small dynamic range (say -12 to -6 dB) or if you choose a big dynamic range (say -40 to -6 dB).

13 Random Durations in a Certain Range

It does not need any explanation how easy it is to add different random durations now. Just remember that each instrument call specifies in the third parameter the duration. This is how the code now looks like: -> I_11.csd

```
if kNextEvent == 1 then
  kPitch random 50, 70
  kVolume random -30, -5
  ;get random duration between 1/3 and 3 seconds ...
  kDuration random 1/3, 3
  ;... and send it to the instrument as p3
  schedulek "PercSine", 0, kDuration, kVolume, kPitch
 endif
```

9 https://csound.com/docs/manual/seed.html

14 Introduce Irregularity

As we want to create a cloud of percussive sine sounds, we certainly must get rid of the regular ticks of the metro. We would like to say something like: "The distance (offset) between two successive notes can be between 1/3 second as minimum and 1 second as maximum."

If we think of *density* as *notes per second*, this means to have a minimum density of 1 note per second, and a maximum density of 3 notes per second. As the density change is valid from one note to the following one, we need to generate it in the moment when one note is being called:

```
kDensity random 1, 3
```

This is actually easy and very much the same as we did for pitch, volume and duration. The somehow tricky point is that we cannot simply write at the beginning of the instrument ...

```
kNextEvent metro kDensity
```

... because *kDensity* has no value here. So we will get the famous Csound error message:

```
error:  Variable 'kDensity' used before defined
```

The solution is to set *kDensity* to an inital value, for instance 2 as mean between 1 and 3:

```
kDensity init 2
```

So this now the cloud code: -> I_12.csd

```
instr Cloud
 ;set the initial density
 kDensity init 1
 ;note that the metro input is now k-rate
 kNextEvent metro kDensity
 ;for each new call of the instrument ...
 if kNextEvent == 1 then
  kPitch random 50, 70
  kVolume random -30, -5
  kDuration random 1/3, 3
  schedulek "PercSine", 0, kDuration, kVolume, kPitch
  ;... generate a new density between 1 and 3 events per second
  kDensity random 1, 3
 endif
endin
```

15 Another Variant in Functional Style

This is a variant for the Cloud instrument in functional style: -> I_12_fs.csd

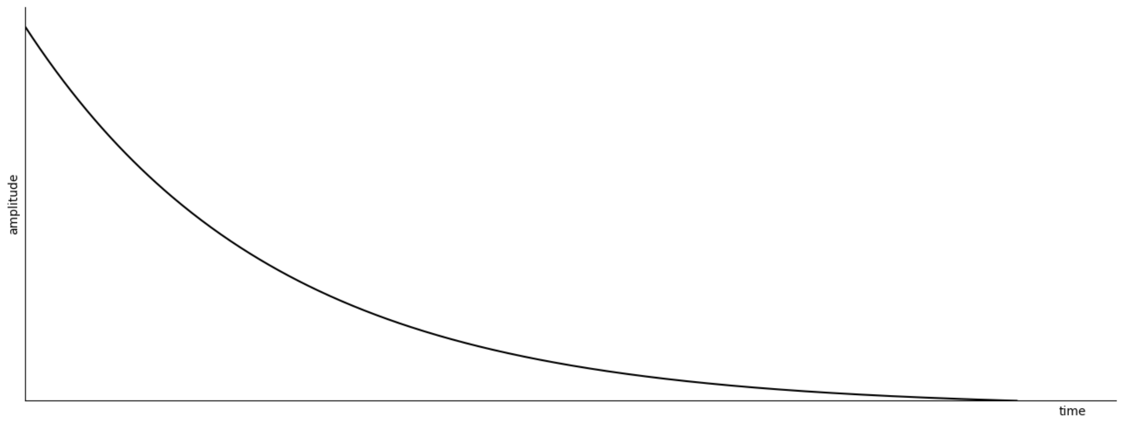
```
instr Cloud
 kDensity init 2
 if(metro:k(kDensity) == 1) then
  schedulek("PercSine",0,random:k(1/3,3),random:k(-30,-5),random:k(50,70))
  kDensity = random:k(1,3)
 endif
endin
```

As we see, we can compress a lot of code. (Whether you prefer it or not is a matter of taste.) But we need the *kDensity* variable, as it is really used as a variable at different places of the code.

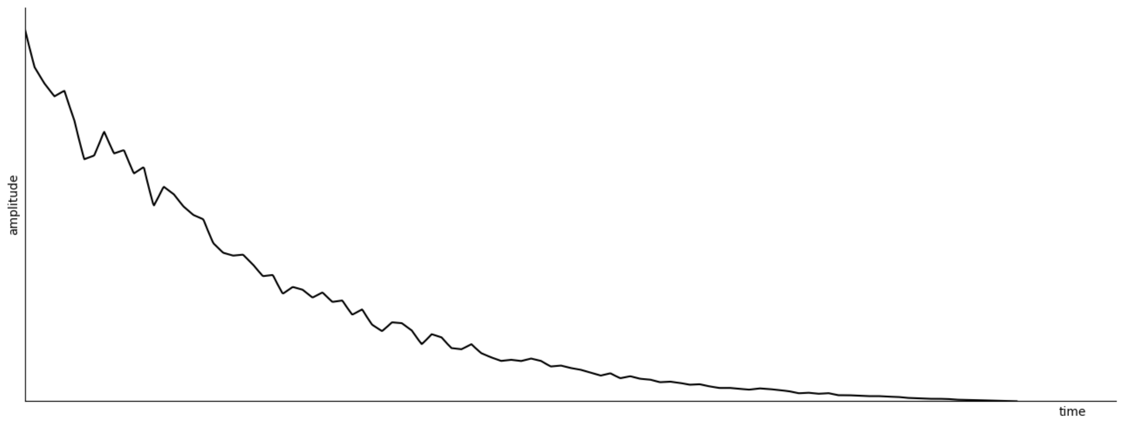
16 Refinements of the Basis Model: Random Deviations

Now as the Cloud instrument is ready for first usages, let us go back to the *PercSine* model of one percussive sine. Compared to natural sounds, or also compared to analogue synthesizers, we have one typical problem in the digital domain: Sounds are "too clean".[10]

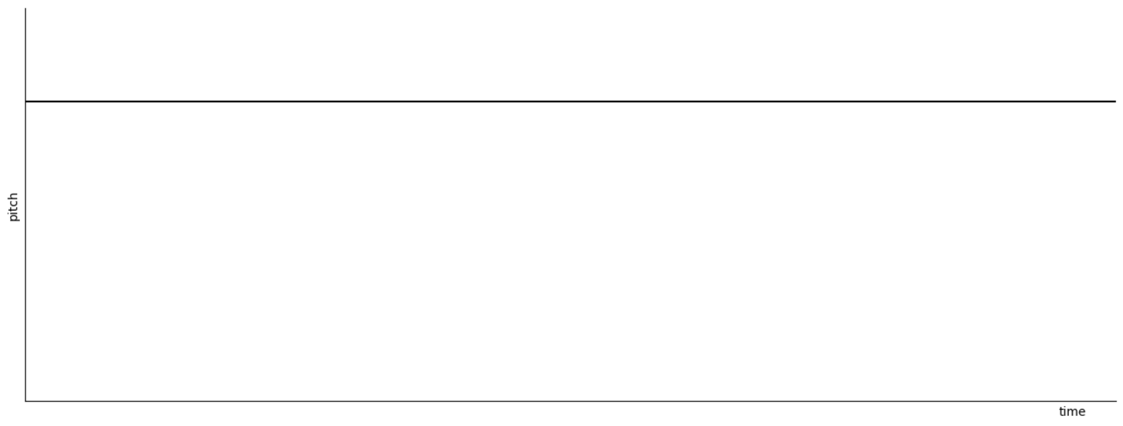
Here, this affects the two main parameters of our percussive sine model: volume and pitch. In nature, we will never see an envelope like this:



Instead, we will see something like this:

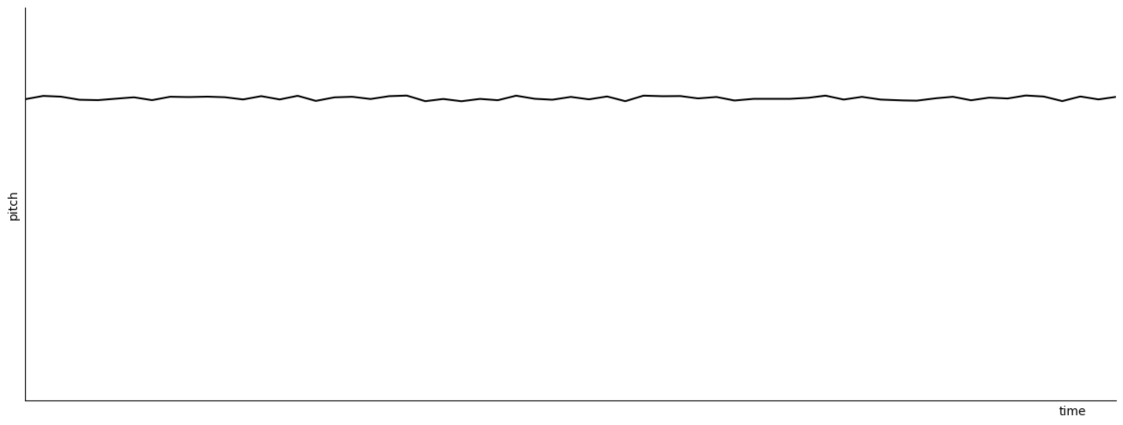


And similar, we will not see a pitch hold by any instrument like this:

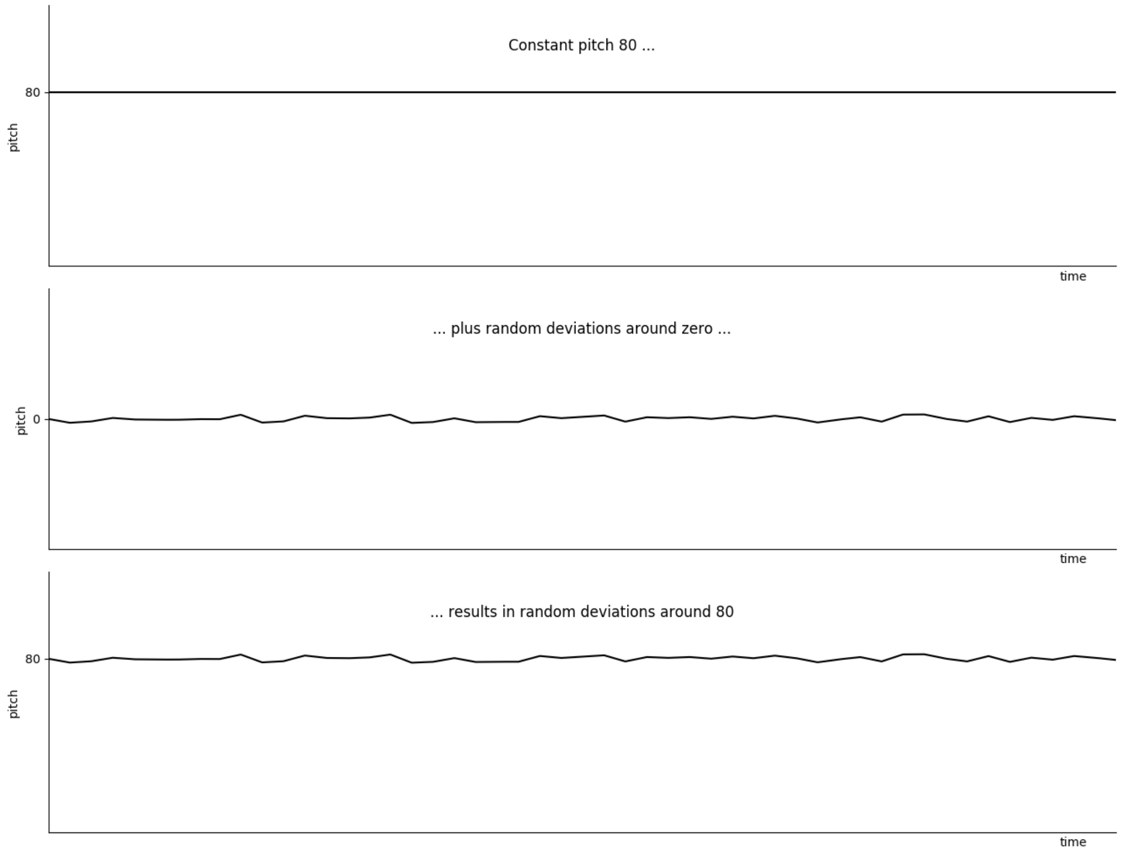


10 I do not mean here that electronic music should mimic or substitute natural sounds. If someone wants to create sounds which are against the normal way of hearing, that is fine. Just remember that our hearing, at least in the physiological dimension, *is* nature.

Instead, we will see something like this:



We can achieve these small fluctuations in sound synthesis, when we add *random deviations* to a straight line:

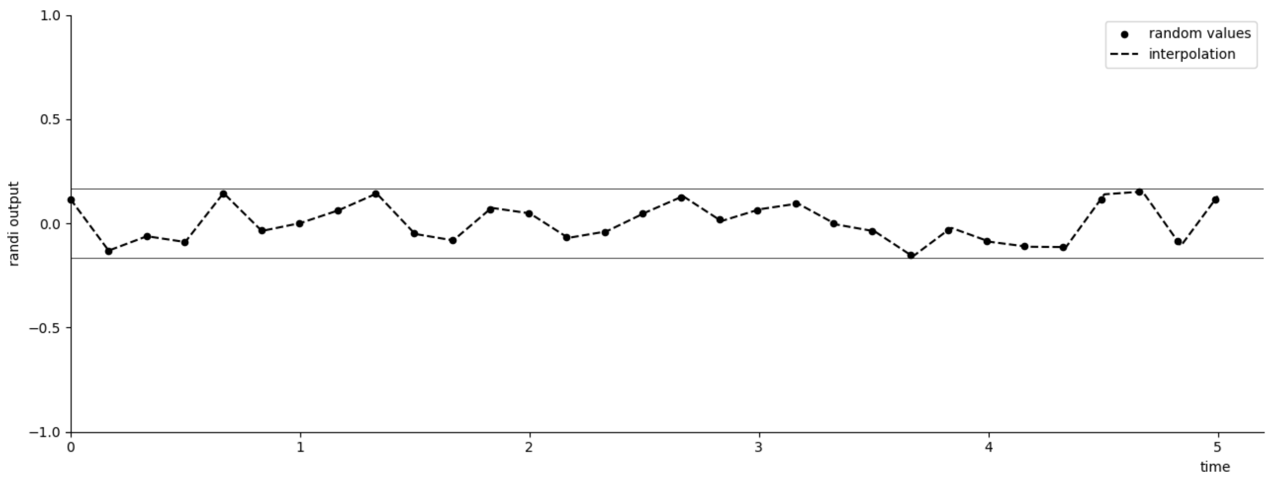


17 Pitch Fluctuations

This is exctly what we need for the pitch fluctuations. We can achieve this "trembling around zero" via the *randi* opcode in Csound. The code line

```
kPchRnd randi 1/6, 6, 2
```

generates a random value between +1/6 and -1/6 in the frequency 6, so six times per second. The values are then connected with each other as straight lines:[11]



Note that the seed is here not set via the global seed, but as third input argument: If greater than 1, it seeds from current time. That is what we usually want, so we set 2 here.

This is the resulting code: I_13.csd

```
instr PercSine
 iDb = -14
 iMidiPitch = 80
 aEnv transeg ampdb(iDb), p3, -4, 0
 ;generate a "trembling" pitch
 kPchRnd randi 1/6, 6, 2
 ;add it to the constant pitch
 aSine poscil aEnv, mtof(iMidiPitch+kPchRnd)
 out aSine, aSine
endin
```

Exercise: Change the range (here 1/6), and change the random frequency (here 6), and listen to the result. Note that the sound gets noisy when the random frequency gets into the audio rate (above 20 Hz).

18 Volume Fluctuations

The volume fluctuations are following the same idea. The code line

```
 aEnvRndDb randi 2, 20, 2
```

generates a random line between +2 and -2 (interpreted as dB) with 20 new points per second. As we have the "clean" envelope as amplitude, not as dB, we must convert the *aEnvRndDb* also to amplitude, and then multiply both: -> I_14.csd

11 This is called *interpolation*, and is the meaning of rand*i* as interpolating random generator.

```
;clean envelopve
aEnv transeg ampdb(iDb), p3, -4, 0
;random deviations ±2dB
aEnvRndDb randi 2, 20, 2
;apply by multiplying
aSine poscil aEnv*ampdb(aEnvRndDb), mtof(iMidiPitch)
```

19 Attack Fluctuations

As last refinement we add some random fluctuation to the attack, by two reasons. Firstly, when we listen carefully to lower pitches (say MIDI note number 48), it sounds a bit noisy. It would be good to have a very short fade-in (in the range of few milliseconds). Secondly, if a percussionist hits a drum or cymbal, we will never have two hits exactly the same. So let us try to implement a random attack time for each percussive sine:

```
iAttackTime random 1/1000, 3/1000 ;time for zero to maximum rise (sec)
aEnv  transeg  0, iAttackTime, 4, ampdb(iDb), p3-iAttackTime, -4, 0
```

We start our *transeg* envelope from zero now and let it rise in *iAttackTime* to the maximum level *ampdb(iDb)*. The shape used for it is the same as for the decay, so 4.[12] We must subtract the *iAttackTime* from the overall duration of the envelope, so we set *p3-iAttackTime* for the second segment.

This is the code now: I_15.csd

```
instr PercSine
 iDb = -14
 iMidiPitch = 70
 iAttackTime random 1/1000, 3/1000 ;rise time from zero to maximum
 aEnv transeg 0, iAttackTime, 4, ampdb(iDb), p3-iAttackTime, -4, 0
 aEnvRndDb randi 2, 20, 2
 kPchRnd randi 1/6, 6, 2
 aSine poscil aEnv*ampdb(aEnvRndDb), mtof(iMidiPitch+kPchRnd)
 out aSine, aSine
endin
```

Exercise: Try different minima / maxima for the attack time and listen to the differences.[13]

20 Clean Up Code

We can go back now to the cloud and insert the new code for the *PercSine* in I_12.csd. And we should write the main cloud parameters as block at the beginning of the instrument definition: The minimum and maximum density, the minimum and maximum pitch, the minimum and maximum volume in dB, and the minimum and maximum duration of a percussive sine.

12 Due to the formula used in the *transeg* opcode rising and decaying segments have different signatures. See the Csound Manual Page for transeg for details and examples.

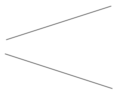
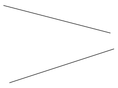
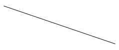
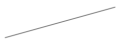
13 I am not perfectly fine with my code here, as actually the attack time should depend on the pitch: lower pitches need a longer attack than higher pitches. But to avoid too sophisticated code, I stick on this here.

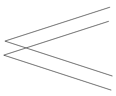
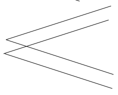
So this is the final result for this first session: -> I_16.csd

```
instr Cloud

 iMinDens = 2
 iMaxDens = 7
 iMinPitch = 66
 iMaxPitch = 80
 iMinVolDb = -40
 iMaxVolDb = -10
 iMinDur = 1/3
 iMaxDur = 3

 iFirstDens = (iMinDens+iMaxDens) / 2
 kDensity init iFirstDens

 kNextEvent metro kDensity
 if kNextEvent == 1 then
  kPitch random iMinPitch, iMaxPitch
  kVolume random iMinVolDb, iMaxVolDb
  kDuration random iMinDur, iMaxDur
  schedulek "PercSine", 0, kDuration, kVolume, kPitch
  kDensity random iMinDens, iMaxDens
 endif
endin

instr PercSine
 iDb = p4
 iMidiPitch = p5
 iAttackTime random 1/1000, 3/1000
 aEnv transeg 0, iAttackTime, 4, ampdb(iDb), p3-iAttackTime, -4, 0
 aEnvRndDb randi 2, 20, 2
 kPchRnd randi 1/10, 6, 2
 aSine poscil aEnv*ampdb(aEnvRndDb), mtof(iMidiPitch+kPchRnd)
 aSine linen aSine, .003, p3, 0
 out aSine, aSine
endin
```

21 Exercises

1. Try the *Cloud* instrument with
   a) long notes and low density
   b) short notes and high density
   How would you describe the musical expression?
   What changes if you put it
   - in different registers (high, middle, low)
   - in small or big ambitus (e.g. minor third versus two octaves)
   - with few or much decibel diferences (try 6, 20, 50)

2. Try to implement these processes:
   a) pitch  (from narrow to wide band)
   b) pitch  (from wide to narrow band)
   c) density  (from high to low density)
   d) density  (from low to high density)

3. Try to implement two processes together:
   a) pitch  (one band rising, one band falling)
   b) duration  (one duration range rising, one duration range falling)
   Which other processes can you combine?
   How much time does a process need?

4. Try to draw sketches of processes, and then try to implement the processes.
   Try to describe which ideas you have but cannot realize with our means.