FAUST

Digital Audio Processing and Synthesis

AGENDA

October 2025

Su	Mo	Tu	We	Th	Fr	Sa
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

November 2025

Su	Mo	Tu	We	Th	Fr	Sa
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30						

Monday, October 13th 2025

02:30PM – 06:30PM (4h)

- ~~Sound~~
- ~~Audio Signal~~
- ~~From Analog to Digital~~
- ~~Sampling~~
- ~~Quantization~~
- ~~Digital Audio Formats~~
- ~~Digital Audio Processing and Synthesis~~
- ~~The Faust programming language~~
- ~~Faust playground!~~

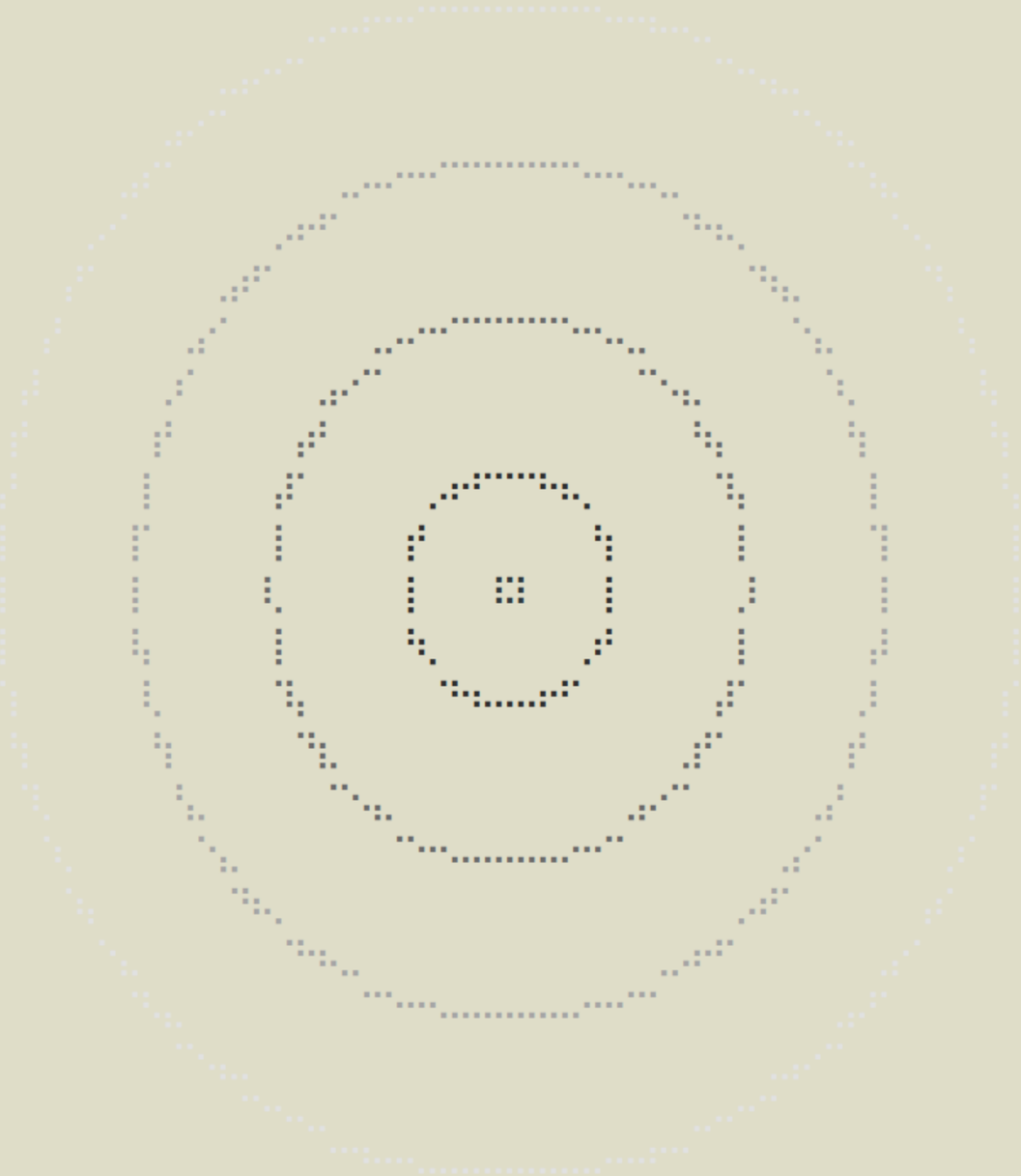
SOUND

[↲] Sound is a ***pressure wave*** that propagates through a **medium** (*gas*, *liquid* or *solid*).

[↲] Propagation is caused by the **oscillation** (*vibration*) of the medium's *particles*, around their ***equilibrium*** positions.

• Sound has the following **properties:**

- **Speed:** ~343 m/s in **air**
- **Amplitude:** in *Pascals* (***Pa***) or *Decibels* (***dB***)
- **Period:** time between two oscillations
- **Wavelength:** distance between two oscillations
- **Frequency:** cycles/sec., in *Hertz* (***Hz***, ***kHz***, ***MHz***)
- **Spectrum**, or *Timbre*



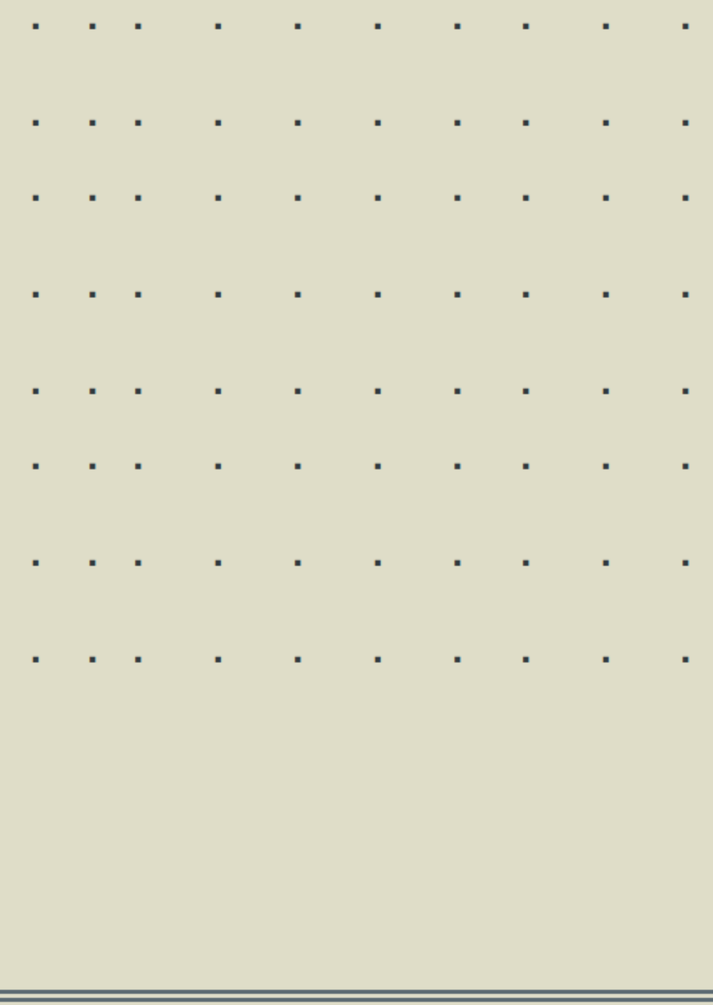
SOUND

[↲] Sound is a ***pressure wave*** that propagates through a **medium** (*gas*, *liquid* or *solid*).

[↲] Propagation is caused by the **oscillation** (*vibration*) of the medium's *particles*, around their ***equilibrium*** positions.

• Sound has the following **properties:**

- **Speed:** ~343 m/s in **air**
- **Amplitude:** in *Pascals* (***Pa***) or *Decibels* (***dB***)
- **Period:** time between two oscillations
- **Wavelength:** distance between two oscillations
- **Frequency:** cycles/sec., in *Hertz* (***Hz***, ***kHz***, ***MHz***)
- **Spectrum**, or *Timbre*

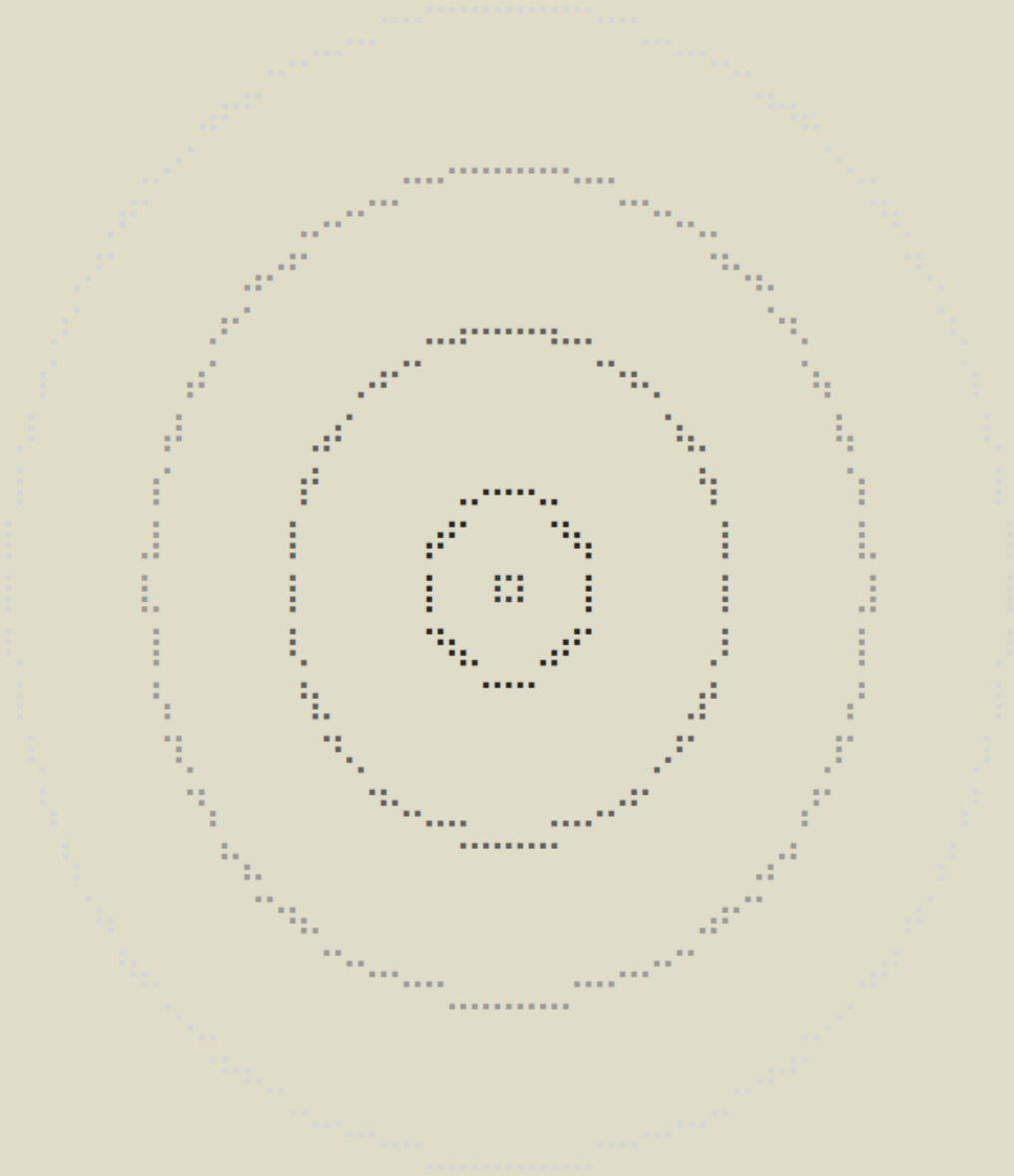


SOUND

[↲] Sound is a ***pressure wave*** that propagates through a **medium** (*gas*, *liquid* or *solid*).

[↲] Propagation is caused by the **oscillation** (*vibration*) of the medium's *particles*, around their ***equilibrium*** positions.

- Sound has the following **properties:**
  - **Speed:** ~343 m/s in **air**
  - **Amplitude:** in *Pascals* (***Pa***) or *Decibels* (***dB***)
  - **Period:** time between two oscillations
  - **Wavelength:** distance between two oscillations
  - **Frequency:** cycles/sec., in *Hertz* (***Hz***, ***kHz***, ***MHz***)
  - **Spectrum**, or *Timbre*



SOUND

• Our ***perception*** of sound is made from the conversion of the vibrations reaching our ***eardrums*** to a *signal of nerve impulses*, transmitted and interpreted by **the brain.**

• Human ears can typically identify sounds ***from 20 Hz to 20 kHz.***

- **Bat:** 2000 to 110,000 Hz
- **Porpoise:** 75 to 150,000 Hz
- **Cat:** 45 to 64,000 Hz
- **Dog:** 67 to 45,000 Hz
- **Chicken:** 125 to 2,000 Hz

SIGNAL

• A ***signal*** describes the evolution of data *over time*. In our case, the vibration of an entity (like the *membrane* of a ***microphone***).

• Just like sound waves turning into nerve impulses, the analyzed data usually needs to be first converted to another *physical unit*, or *domain* (***transduction***) in order to adapt to measurement/processing tools.

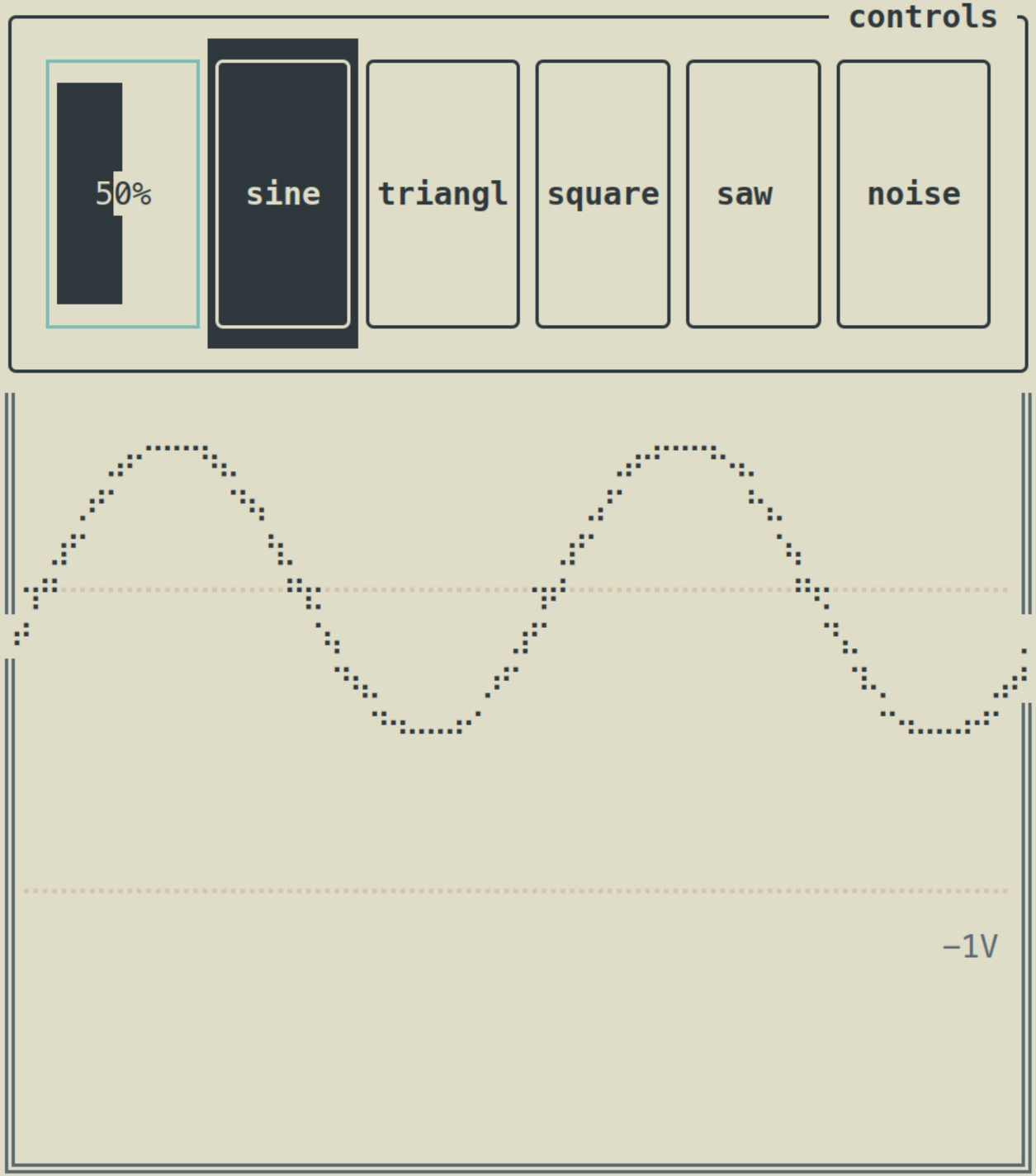
• The vibration of a microphone's membrane is, for instance, usually converted to ***continuous electrical current***, before it can be processed and/or analyzed. In this case, the signal is said to be "***analog***".

SIGNAL

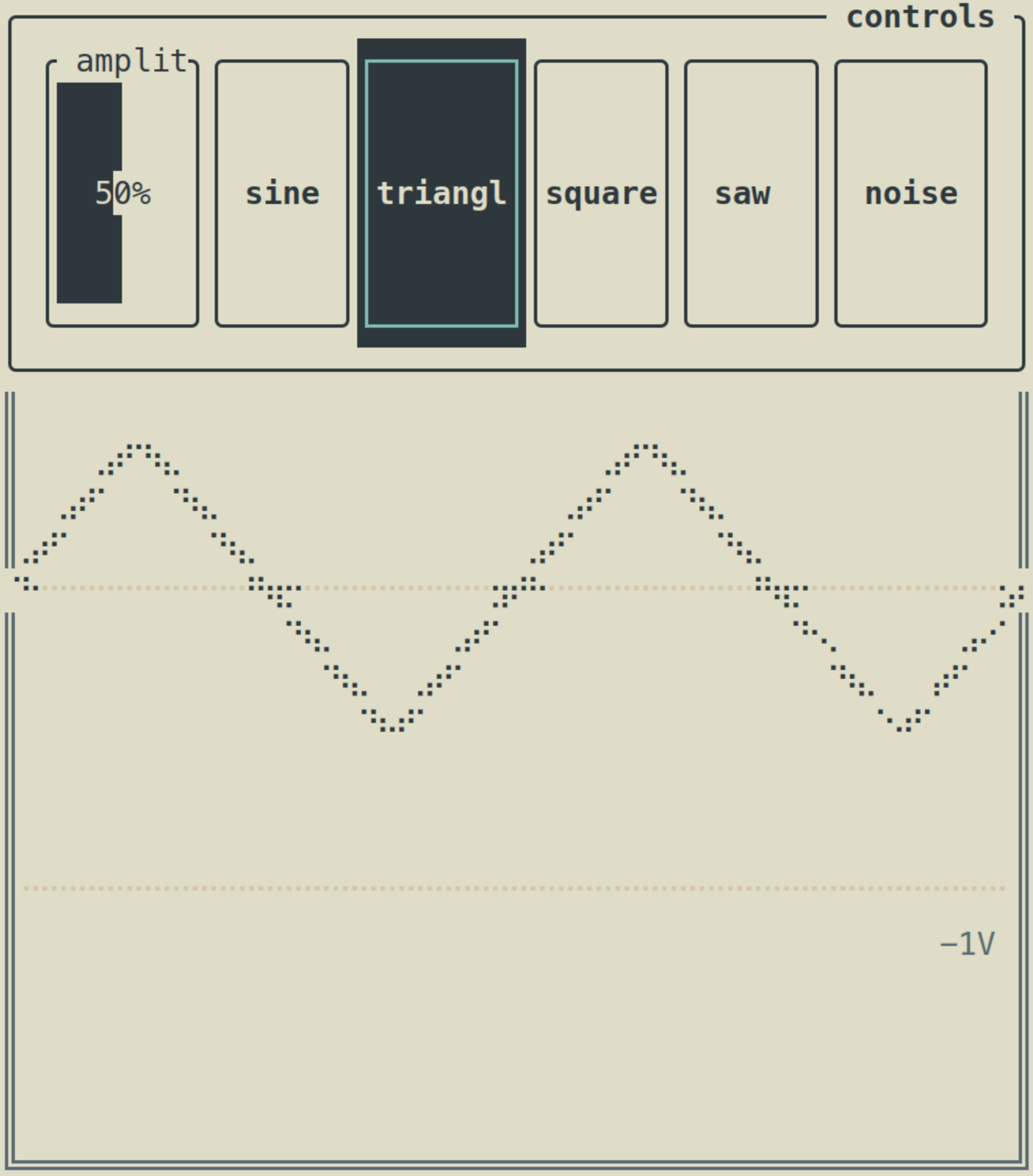
[↲] With an oscilloscope, we can measure the **amplitude** of a signal at a given *point in time* (***time-domain***), through the visualisation of a ***waveform***.

• An analog signal can already be processed as it is, with ***analog effects***: *tape delay*, *distortion*, *chorus*, *reverberation (spring, plate)*, etc.

[↲] On the other hand, it is difficult to extract precise information about ***frequency*** and ***spectrum***. For this purpose, it's far more efficient to switch to the ***frequency domain***, which requires the analog signal to be turned into a ***digital signal***...



SIGNAL



[↲] With an oscilloscope, we can measure the **amplitude** of a signal at a given *point in time* (***time-domain***), through the visualisation of a ***waveform***.

• An analog signal can already be processed as it is, with ***analog effects***: *tape delay, distortion, chorus, reverberation (spring, plate)*, etc.

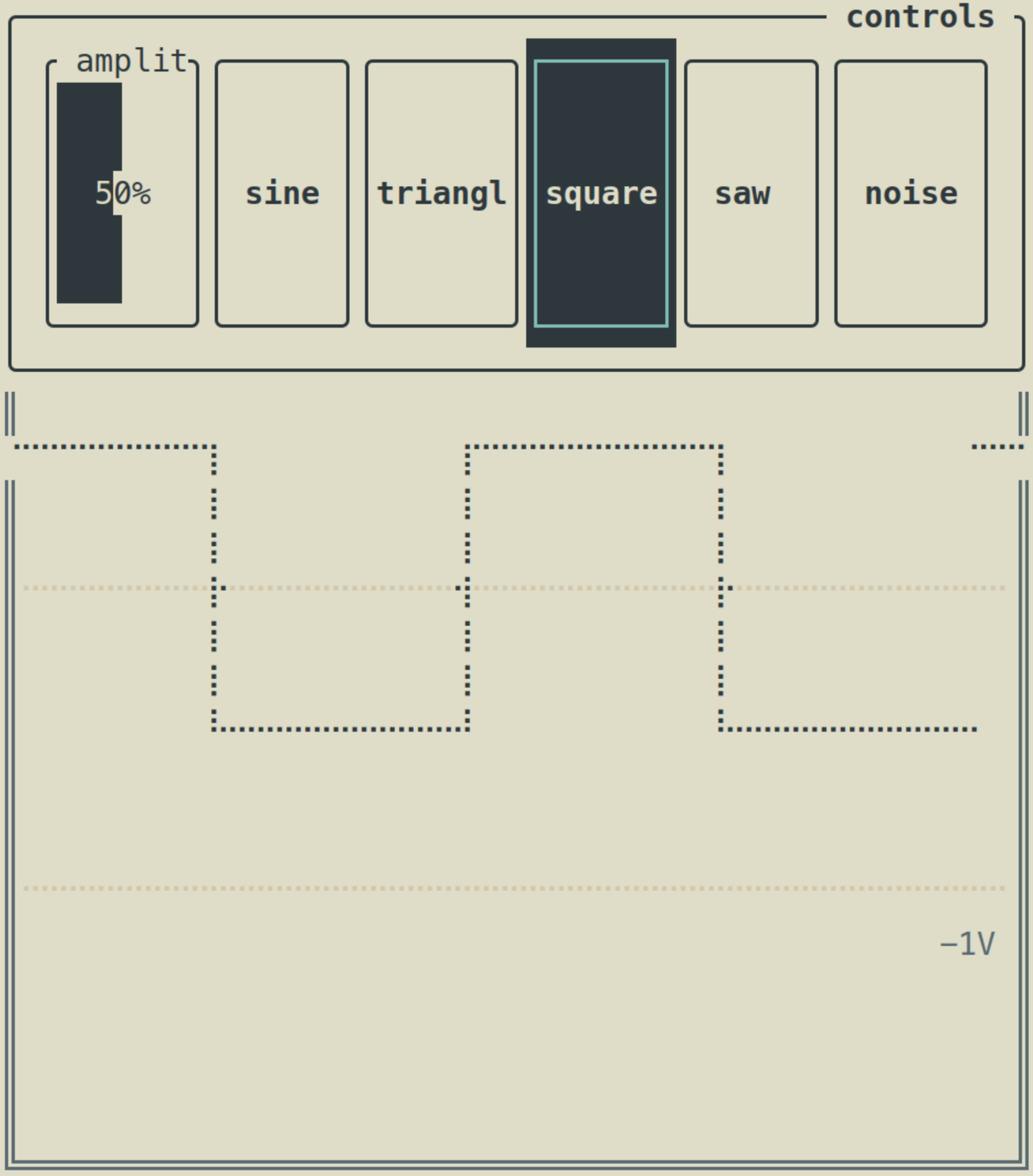
[↲] On the other hand, it is difficult to extract precise information about ***frequency*** and ***spectrum***. For this purpose, it's far more efficient to switch to the ***frequency domain***, which requires the analog signal to be turned into a ***digital signal***...

SIGNAL

[↲] With an oscilloscope, we can measure the **amplitude** of a signal at a given *point in time* (***time-domain***), through the visualisation of a ***waveform***.

• An analog signal can already be processed as it is, with ***analog effects***: *tape delay, distortion, chorus, reverberation (spring, plate)*, etc.

[↲] On the other hand, it is difficult to extract precise information about ***frequency*** and ***spectrum***. For this purpose, it's far more efficient to switch to the ***frequency domain***, which requires the analog signal to be turned into a ***digital signal***...

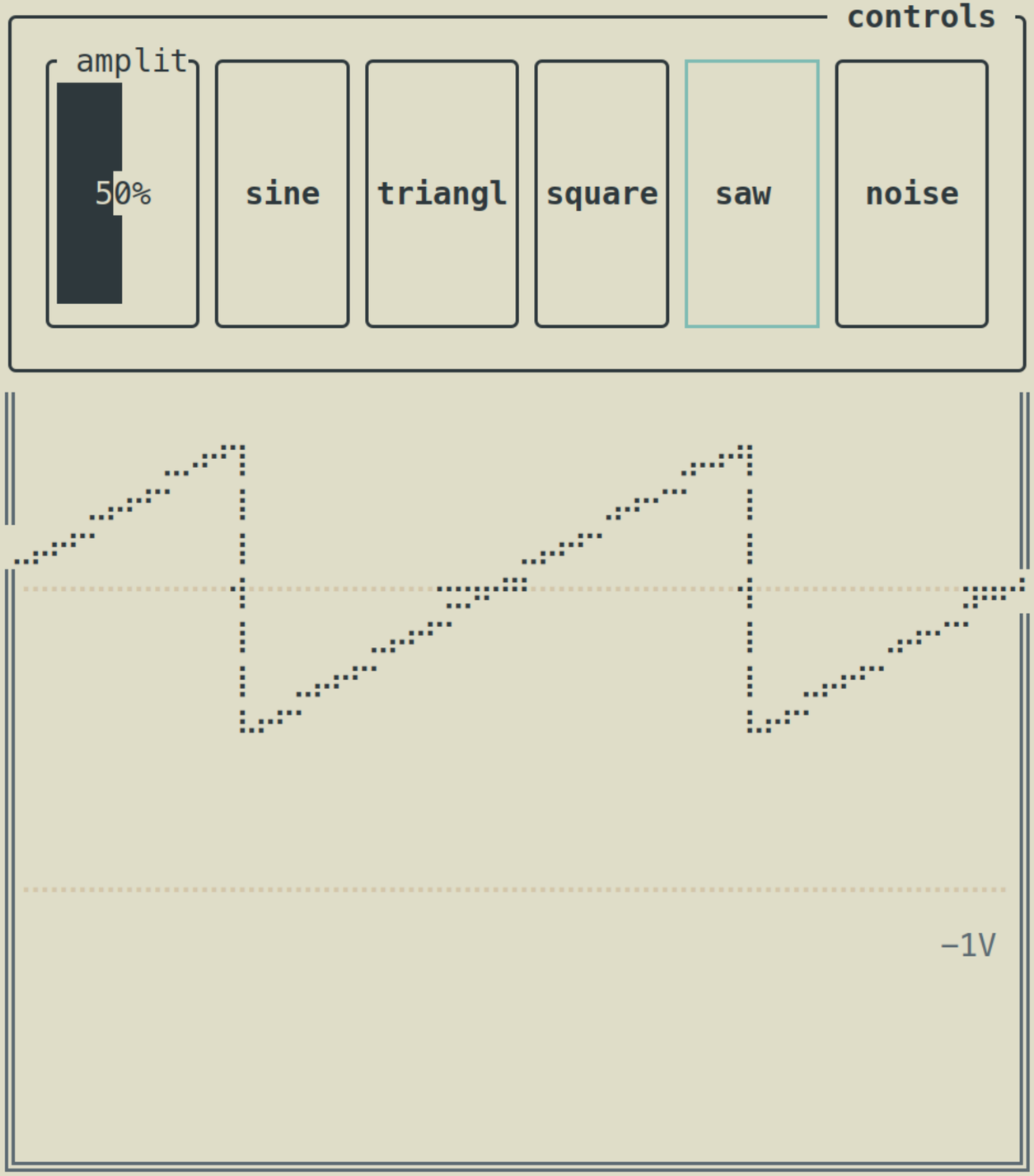


SIGNAL

[↲] With an oscilloscope, we can measure the **amplitude** of a signal at a given *point in time* (***time-domain***), through the visualisation of a ***waveform***.

• An analog signal can already be processed as it is, with ***analog effects***: *tape delay*, *distortion*, *chorus*, *reverberation (spring, plate)*, etc.

[↲] On the other hand, it is difficult to extract precise information about ***frequency*** and ***spectrum***. For this purpose, it's far more efficient to switch to the ***frequency domain***, which requires the analog signal to be turned into a ***digital signal***...



SIGNAL

[↲] With an oscilloscope, we can measure the **amplitude** of a signal at a given *point in time* (***time-domain***), through the visualisation of a ***waveform***.

• An analog signal can already be processed as it is, with ***analog effects***: *tape delay, distortion, chorus, reverberation (spring, plate)*, etc.

[↲] On the other hand, it is difficult to extract precise information about ***frequency*** and ***spectrum***. For this purpose, it's far more efficient to switch to the ***frequency domain***, which requires the analog signal to be turned into a ***digital signal***...

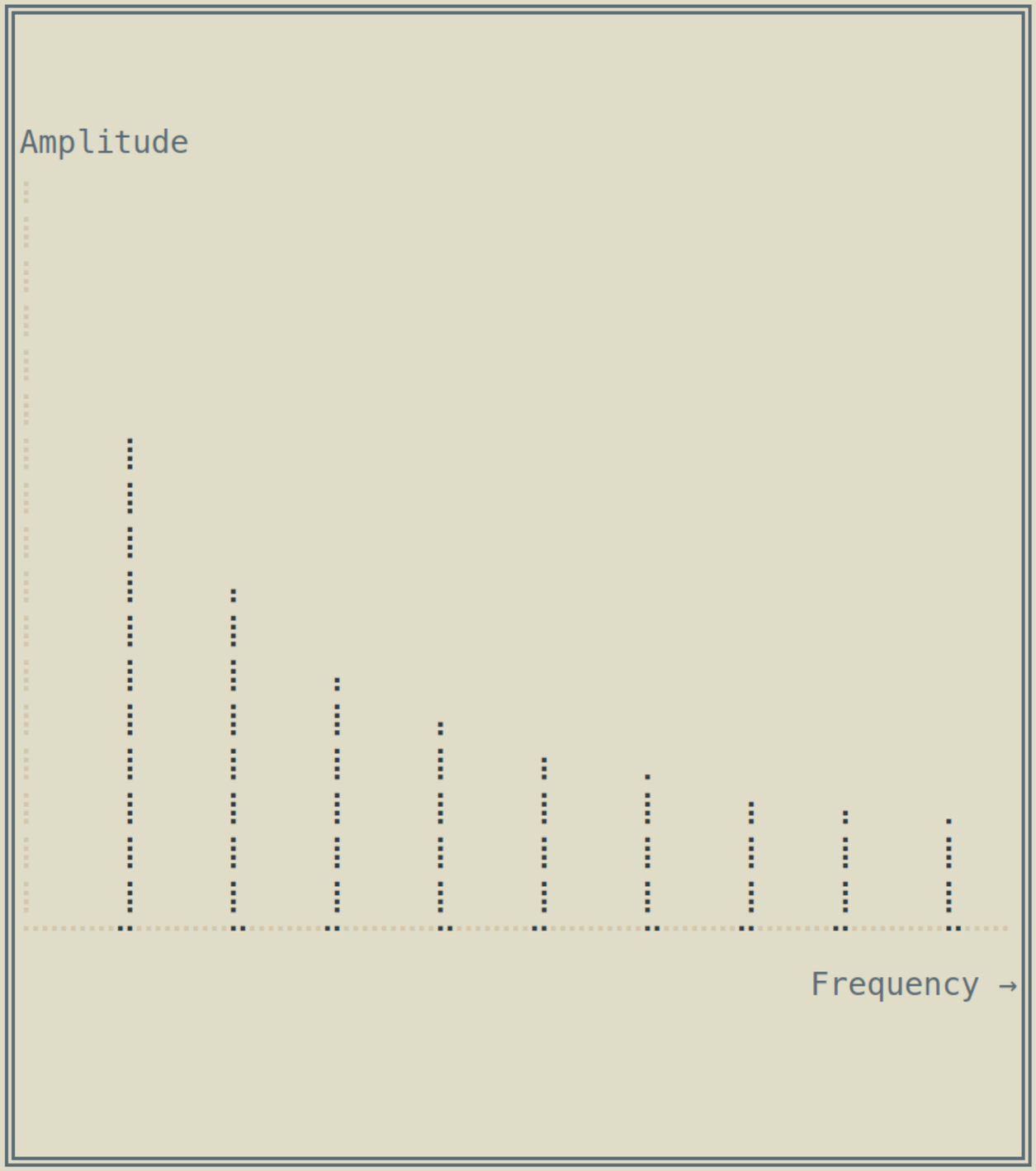
@pchdev – **Emeraude** – *Inria, INSA-Lyon, CITI Lab*

SIGNAL

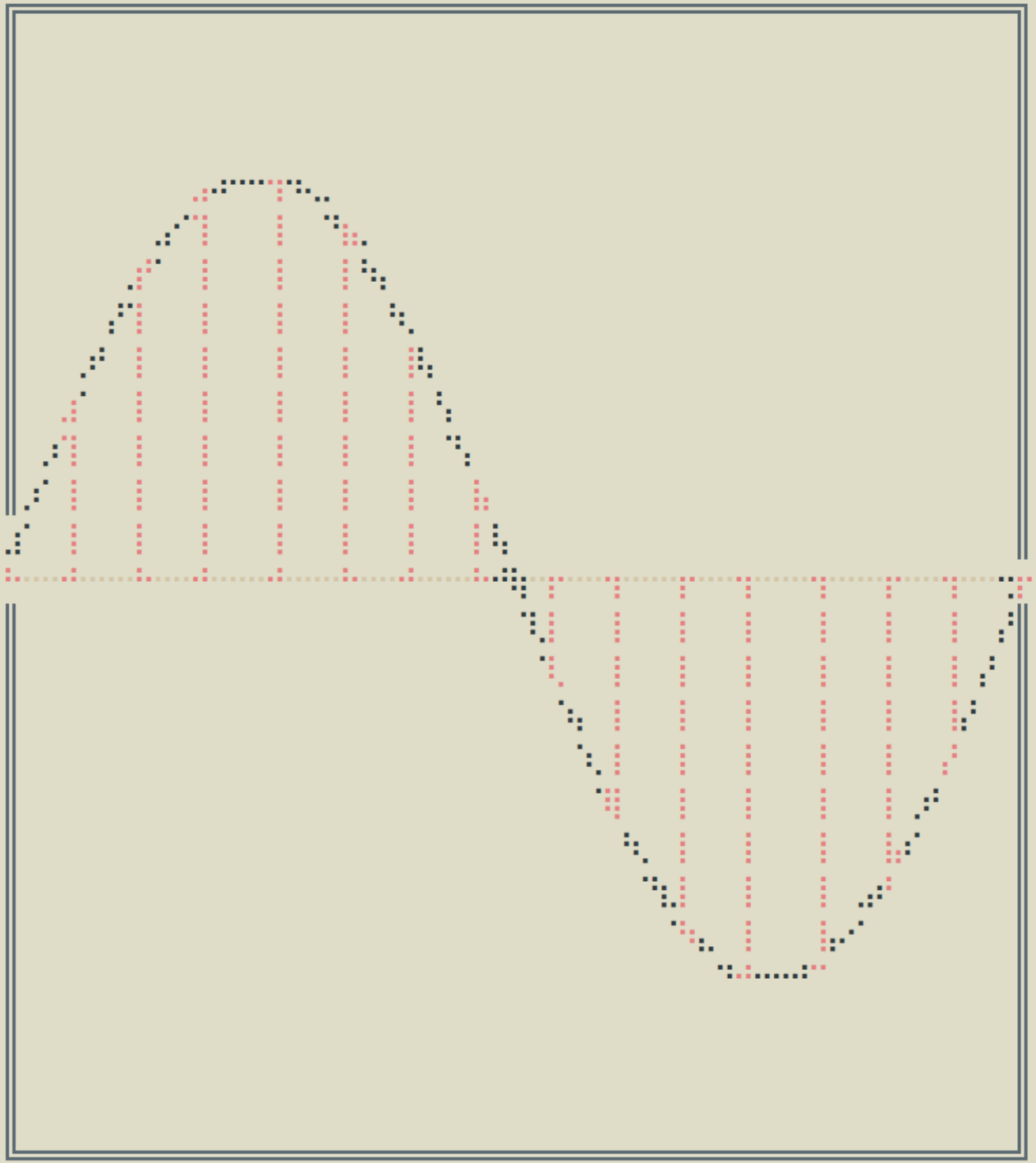
[↲] With an oscilloscope, we can measure the **amplitude** of a signal at a given *point in time* (***time-domain***), through the visualisation of a ***waveform***.

• An analog signal can already be processed as it is, with ***analog effects***: *tape delay*, *distortion*, *chorus*, *reverberation (spring, plate)*, etc.

[↲] On the other hand, it is difficult to extract precise information about ***frequency*** and ***spectrum***. For this purpose, it's far more efficient to switch to the ***frequency domain***, which requires the analog signal to be turned into a ***digital signal***...



DIGITAL SIGNAL



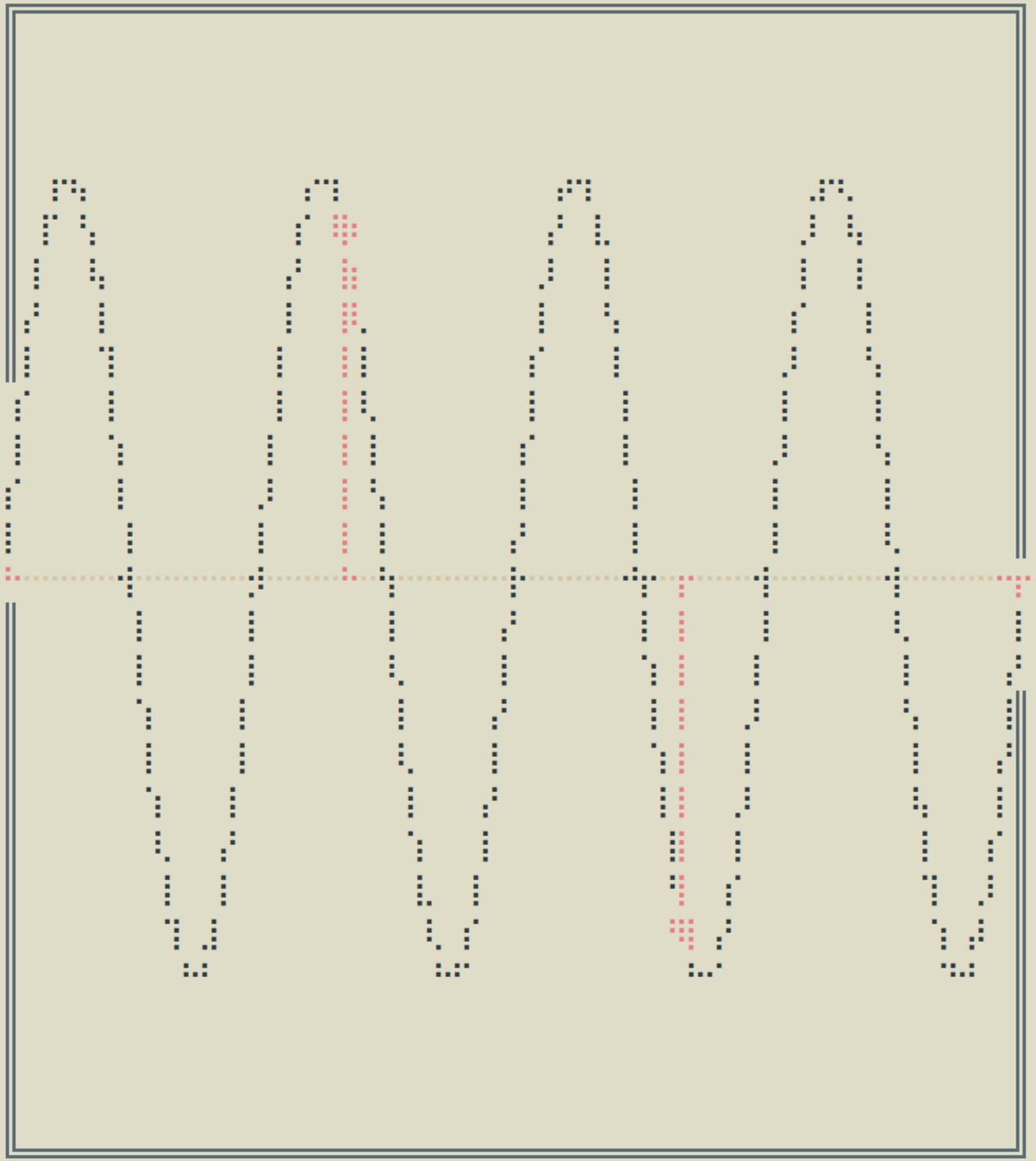
• To ***digitize*** a continuous signal implies *discretizing* it. This is made possible by an *Analog-to-Digital Conversion* (**ADC**) process, which implies two key elements: ***sampling*** and ***quantization.***

[↲] ***Sampling*** means taking a sample of a signal at a certain frequency/rate (***sample rate***).

- **Audio CD:** *44.1 kHz*
- **Pro Audio:** *48/96 kHz*
- **MP3:** *320/256/128/96/64* **kbps**

[↲] • Because of ***aliasing***, the *sampling rate* must be **at least two times superior** to the **highest frequency** we want to represent (*Nyquist-Shannon*).

DIGITAL SIGNAL



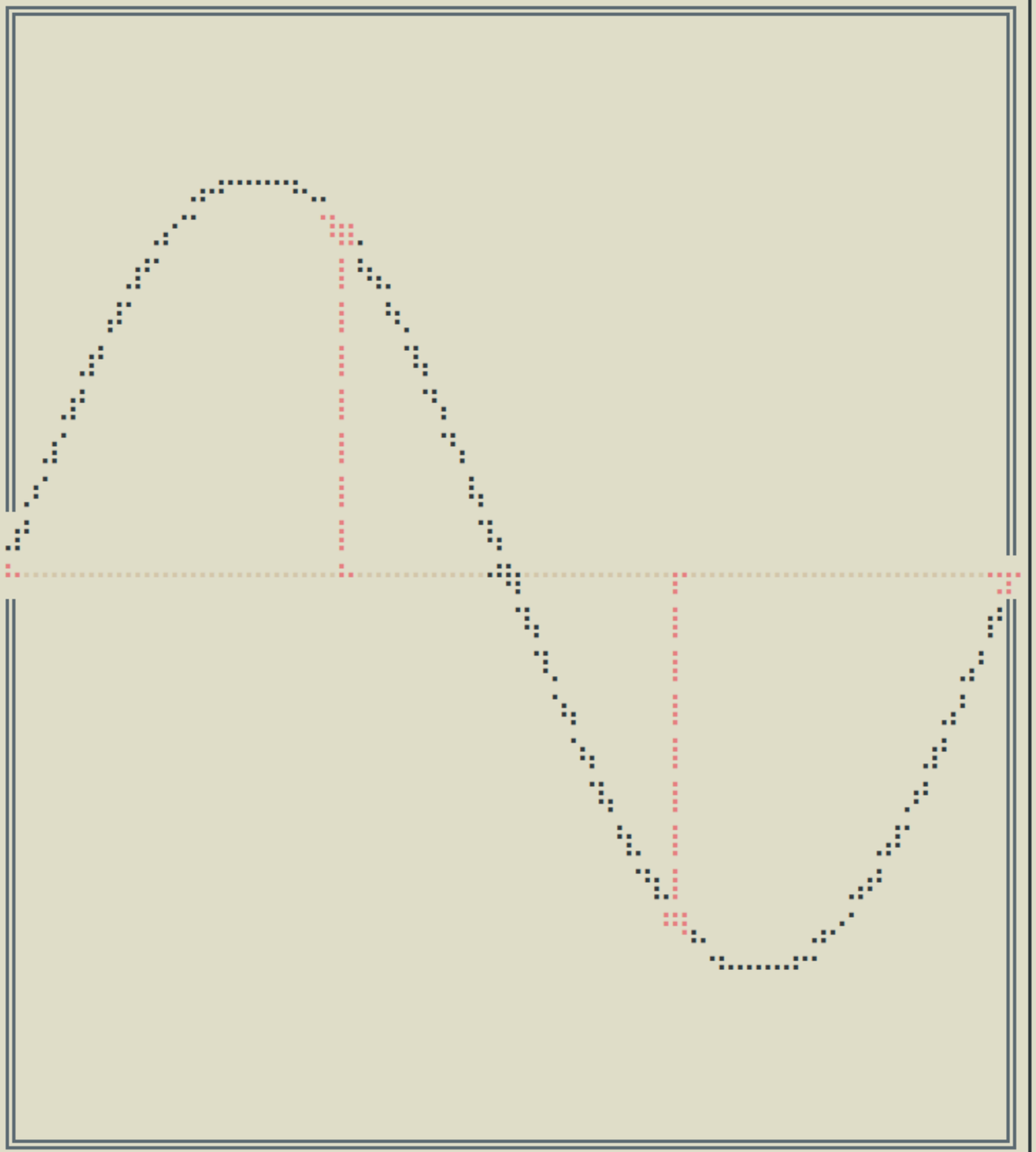
• To ***digitize*** a continuous signal implies *discretizing* it. This is made possible by an *Analog-to-Digital Conversion* (**ADC**) process, which implies two key elements: ***sampling*** and ***quantization.***

[↲] ***Sampling*** means taking a sample of a signal at a certain frequency/rate (***sample rate***).

- **Audio CD:** *44.1 kHz*
- **Pro Audio:** *48/96 kHz*
- **MP3:** *320/256/128/96/64* **kbps**

[↲] • Because of ***aliasing***, the *sampling rate* must be **at least two times superior** to the **highest frequency** we want to represent (*Nyquist-Shannon*).

DIGITAL SIGNAL



• To ***digitize*** a continuous signal implies *discretizing* it. This is made possible by an *Analog-to-Digital Conversion* (**ADC**) process, which implies two key elements: ***sampling*** and ***quantization.***

[↲] ***Sampling*** means taking a sample of a signal at a certain frequency/rate (***sample rate***).

- **Audio CD:** *44.1 kHz*
- **Pro Audio:** *48/96 kHz*
- **MP3:** *320/256/128/96/64* **kbps**

[↲] • Because of ***aliasing***, the *sampling rate* must be **at least two times superior** to the **highest frequency** we want to represent (*Nyquist-Shannon*).

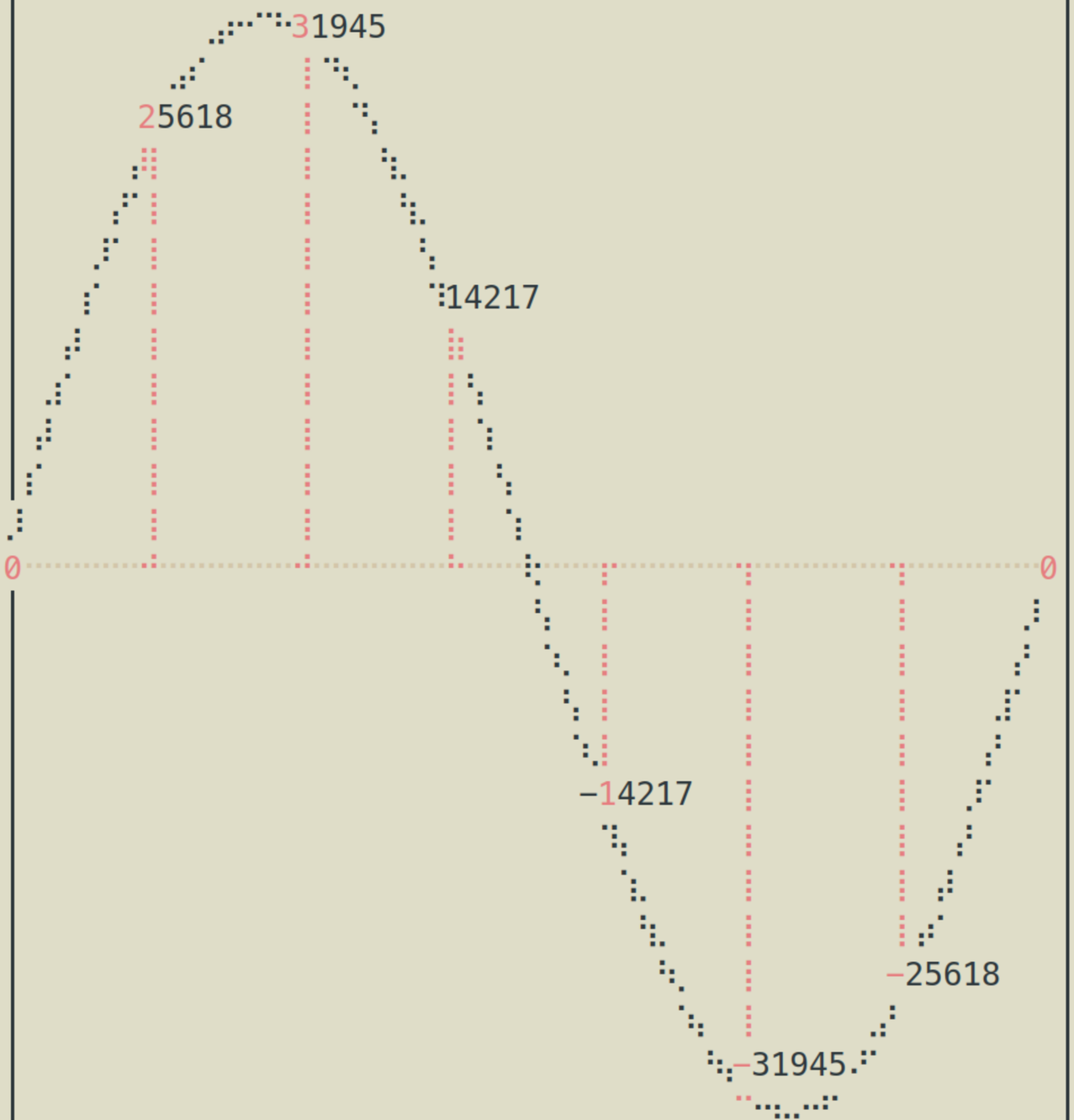
DIGITAL SIGNAL

[↲] • Once we take a sample of a signal at a given time, we need to determine the ***scale of its value***, this is called ***quantization***. Increasing the scale implies reducing the ***quantization noise*** (*quality vs. storage tradeoff*).

↳ • **Audio CD**: 16-bits (65,536 values, 98 dB SNR)
  • **Pro Audio**: 24-bits (~16,7 mil., 146 dB SNR)
  • ***DSP***: 32/64-bits float (~4,3 bil., 194 dB SNR)

• For **DSP**, it is easier to make computations in ***floating-point*** (decimal), and *normalize* the signal between *-1.0* and *1.0*.

• Finally, sending a digital signal to audio speakers involves the inverse process of an **ADC**: *Digital-to-Analog Conversion* (**DAC**).



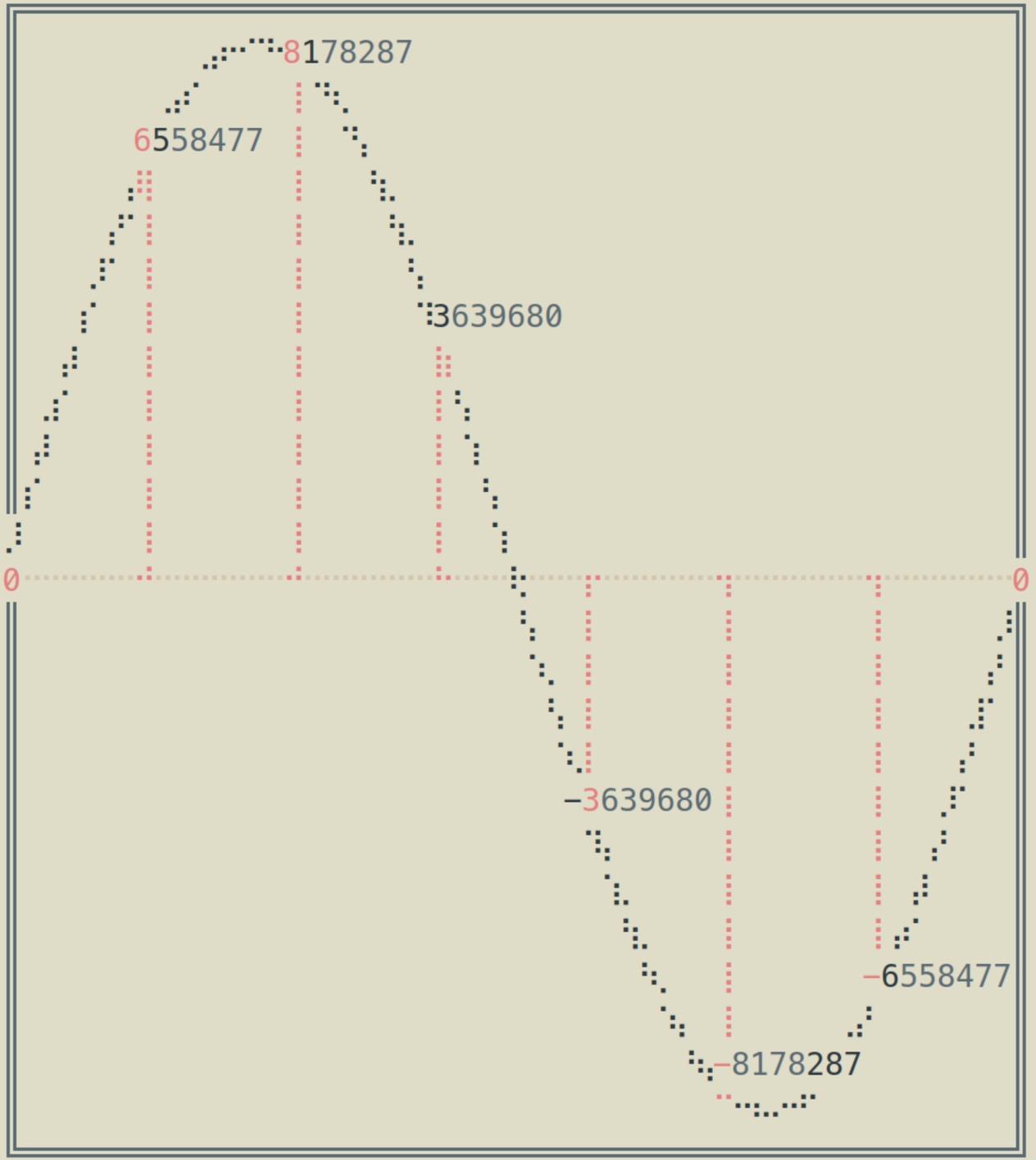
DIGITAL SIGNAL

[↲] • Once we take a sample of a signal at a given time, we need to determine the ***scale of its value***, this is called ***quantization***. Increasing the scale implies reducing the ***quantization noise*** (*quality vs. storage tradeoff*).

- **Audio CD**: 16-bits (65,536 values, 98 dB SNR)
- **Pro Audio**: 24-bits (~16,7 mil., 146 dB SNR)
- ***DSP***: 32/64-bits float (~4,3 bil., 194 dB SNR)

• For **DSP**, it is easier to make computations in ***floating-point*** (decimal), and *normalize* the signal between *-1.0* and *1.0*.

• Finally, sending a digital signal to audio speakers involves the inverse process of an **ADC**: *Digital-to-Analog Conversion* (**DAC**).



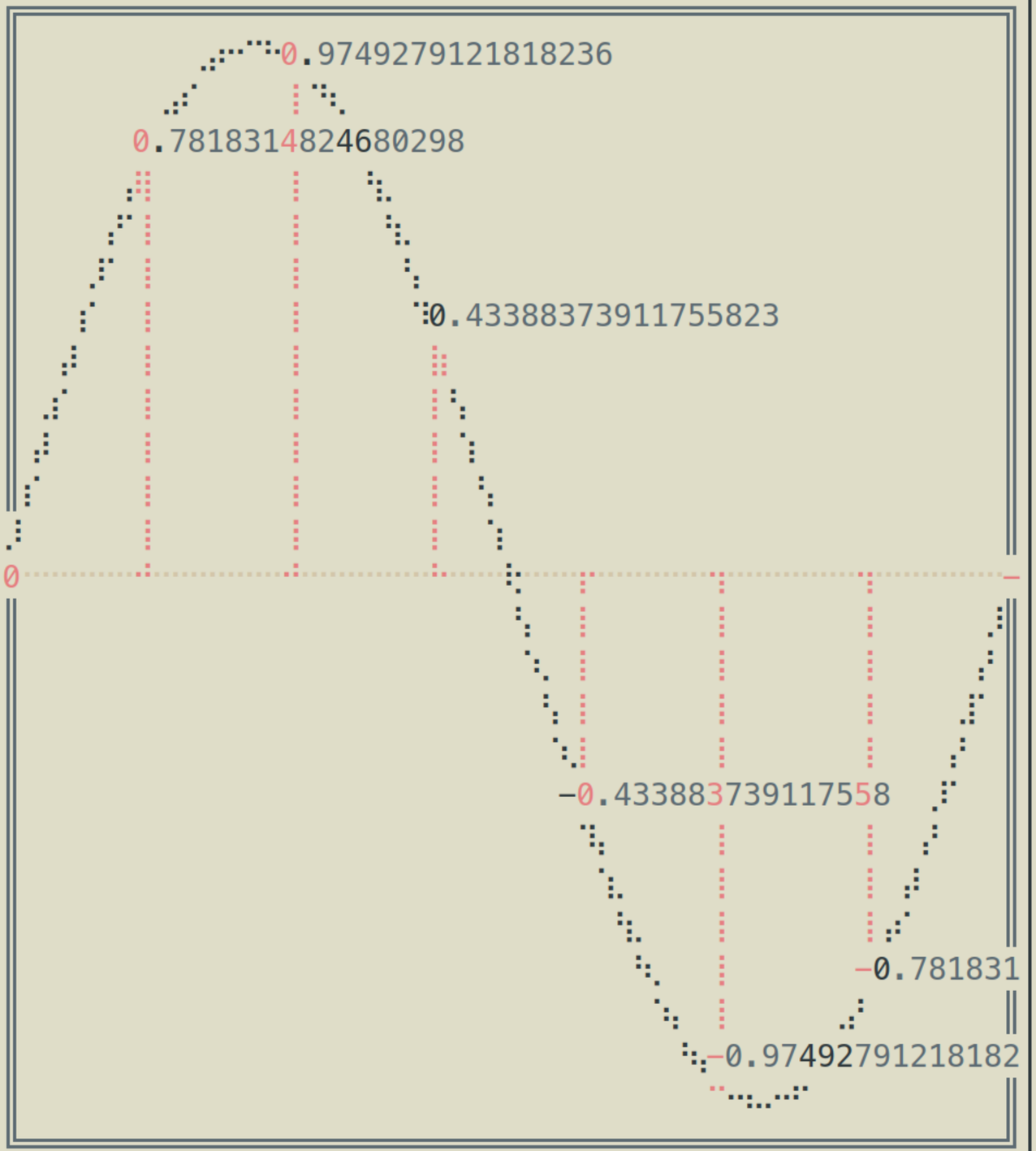
DIGITAL SIGNAL

[↲] • Once we take a sample of a signal at a given time, we need to determine the ***scale of its value***, this is called ***quantization***. Increasing the scale implies reducing the ***quantization noise*** (*quality vs. storage tradeoff*).

- **Audio CD**: 16-bits (65,536 values, 98 dB SNR)
- **Pro Audio**: 24-bits (~16,7 mil., 146 dB SNR)
- ***DSP***: 32/64-bits float (~4,3 bil., 194 dB SNR)

• For **DSP**, it is easier to make computations in ***floating-point*** (decimal), and *normalize* the signal between *-1.0* and *1.0*.

• Finally, sending a digital signal to audio speakers involves the inverse process of an **ADC**: *Digital-to-Analog Conversion* (**DAC**).



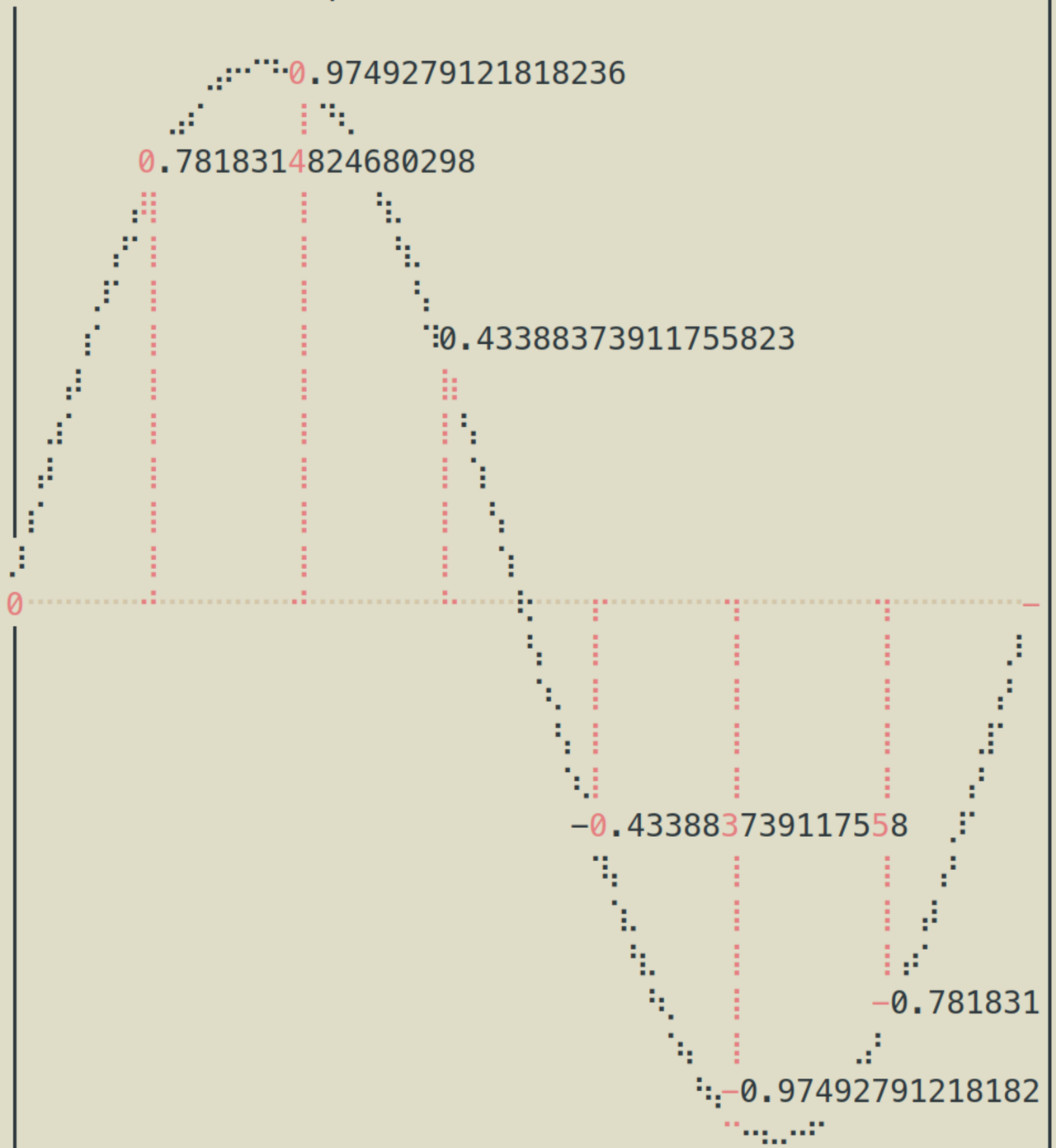
DIGITAL SIGNAL

[↲] • Once we take a sample of a signal at a given time, we need to determine the ***scale of its value***, this is called ***quantization***. Increasing the scale implies reducing the ***quantization noise*** (*quality vs. storage tradeoff*).

- **Audio CD**: 16-bits (65,536 values, 98 dB SNR)
- **Pro Audio**: 24-bits (~16,7 mil., 146 dB SNR)
- ***DSP***: 32/64-bits float (~4,3 bil., 194 dB SNR)

• For **DSP**, it is easier to make computations in ***floating-point*** (decimal), and *normalize* the signal between *-1.0* and *1.0*.

• Finally, sending a digital signal to audio speakers involves the inverse process of an **ADC**: *Digital-to-Analog Conversion* (**DAC**).



ENTER FAUST

[↲] **Faust** (*Functional Audio Stream*) is a programming language specifically made for ***audio DSP and synthesis***. It was created by **Yann Orlarey, Dominique Fober** & **Stéphane Letz** at **GRAME** in 2002.

+ *Functional* paradigm
+ Declarative, math-like syntax
+ Produces optimized code for many architectures
- Not recommended (yet) for multi-rate (FFT)

- **Plugins:** VST, CLAP, AudioUnit
- **Software:** Max, Pd, SuperCollider, Csound
- **OS:** Linux, macOS, Windows, Android, iOS
- **Embedded:** Bela, Teensy, Daisy, ESP32, FPGA
- **Code:** C/C++, Rust, WASM, Java...

[↲] Tutorials & documentation available at:
https://faust.grame.fr

[↲] Dedicated online IDE:
https://faustide.grame.fr

@pchdev – **Emeraude** – *Inria*, *INSA-Lyon*, *CITI Lab*

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

process = os.oscrc(440) * 0.25;
```

Normal

BASICS

[↲] Basic program: ***import*** and ***process*** statements.

[↲] Use a ***Sinewave Oscillator*** from the library.

[↲] In DSP, we represent an audio signal with ***floating-point*** (decimal) numbers and *scale* its values ***between -1.0 and +1.0***.

[↲] ***Mixing*** two signals can be done using the '+' operator.

[↲] When multiplying two signals together, we can already make a simple DSP effect: ***Amplitude Modulation*** (or ***Ring Modulation***).

[↲] Faust can process **multiple channels**, we can use the ',' symbol (***Parallel Operator***) to put signals *in parallel*.

[↲] We can declare **custom variables** (***functions***).

editor control graph wave spectrum

faust

```
// Import statement:
// we import all of the faust libraries.
import("stdfaust.lib");

// 'process' statement:
// what your program will be processing/running:
process = 0;

// /!\ IMPORTANT:
// All the code expressions in Faust
// end with a semicolon ';'
```

Normal

BASICS

[↲] Basic program: ***import*** and ***process*** statements.

[↲] Use a ***Sinewave Oscillator*** from the library.

[↲] In DSP, we represent an audio signal with ***floating-point*** (decimal) numbers and *scale* its values ***between -1.0 and +1.0***.

[↲] ***Mixing*** two signals can be done using the '+' operator.

[↲] When multiplying two signals together, we can already make a simple DSP effect: ***Amplitude Modulation*** (or ***Ring Modulation***).

[↲] Faust can process **multiple channels**, we can use the ',' symbol (***Parallel Operator***) to put signals *in parallel*.

[↲] We can declare **custom variables** (***functions***).

editor control graph wave spectrum

faust

```
// Import statement:
// we import all of the faust libraries.
import("stdfaust.lib");

// Here, we use the 'oscrc' function
// from the 'os' (oscillators) library:
process = os.oscrc(440) * 0.25;
```

Normal

BASICS

[↲] Basic program: ***import*** and ***process*** statements.

[↲] Use a ***Sinewave Oscillator*** from the library.

[↲] In DSP, we represent an audio signal with ***floating-point*** (decimal) numbers and *scale* its values ***between -1.0 and +1.0***.

[↲] ***Mixing*** two signals can be done using the '+' operator.

[↲] When multiplying two signals together, we can already make a simple DSP effect: ***Amplitude Modulation*** (or ***Ring Modulation***).

[↲] Faust can process **multiple channels**, we can use the ',' symbol (***Parallel Operator***) to put signals *in parallel*.

[↲] We can declare **custom variables** (***functions***).

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

// We run the oscillators at 440Hz (A440),
// we multiply its amplitude by 0.25,
// which means we divide its volume by 4:
process = os.oscrc(440) * 0.25;

// Which would be the same as:
process = os.oscrc(440) / 4;
```

Normal

BASICS

[↲] Basic program: ***import*** and ***process*** statements.

[↲] Use a ***Sinewave Oscillator*** from the library.

[↲] In DSP, we represent an audio signal with ***floating-point*** (decimal) numbers and *scale* its values ***between -1.0 and +1.0***.

[↲] ***Mixing*** two signals can be done using the '+' operator.

[↲] When multiplying two signals together, we can already make a simple DSP effect: ***Amplitude Modulation*** (or ***Ring Modulation***).

[↲] Faust can process **multiple channels**, we can use the ',' symbol (***Parallel Operator***) to put signals *in parallel*.

[↲] We can declare **custom variables** (***functions***).

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

// Mixing two signals means adding them together,
// which can be done by using the '+' operator
process = (os.oscrc(440) + os.oscrc(880)) * 0.25;

// Here, we used parentheses because of
// operator precedence.
// Multiplications are always evaluated before
// additions or substractions.
```

Normal

BASICS

[↲] Basic program: ***import*** and ***process*** statements.

[↲] Use a ***Sinewave Oscillator*** from the library.

[↲] In DSP, we represent an audio signal with ***floating-point*** (decimal) numbers and *scale* its values ***between -1.0 and +1.0***.

[↲] ***Mixing*** two signals can be done using the '+' operator.

[↲] When multiplying two signals together, we can already make a simple DSP effect: ***Amplitude Modulation*** (or ***Ring Modulation***).

[↲] Faust can process **multiple channels**, we can use the ',' symbol (***Parallel Operator***) to put signals *in parallel*.

[↲] We can declare **custom variables** (***functions***).

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

// Basic amplitude modulation.
// no need for parentheses here, since we only
// use multiplications:
process = os.oscrc(440) * os.oscrc(10) * 0.25;
```

Normal

BASICS

[↲] Basic program: ***import*** and ***process*** statements.

[↲] Use a ***Sinewave Oscillator*** from the library.

[↲] In DSP, we represent an audio signal with ***floating-point*** (decimal) numbers and *scale* its values ***between -1.0 and +1.0***.

[↲] ***Mixing*** two signals can be done using the '**+**' operator.

[↲] When multiplying two signals together, we can already make a simple DSP effect: ***Amplitude Modulation*** (or ***Ring Modulation***).

[↲] Faust can process **multiple channels**, we can use the '**,**' symbol (***Parallel Operator***) to put signals *in parallel*.

[↲] We can declare **custom variables** (***functions***).

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

// Here, we use the Parallel Operator ','
// to put two signals in parallel:
process = (os.oscrc(440) * 0.25), (os.oscrc(880) *
0.25);
```

Normal

BASICS

[↲] Basic program: ***import*** and ***process*** statements.

[↲] Use a ***Sinewave Oscillator*** from the library.

[↲] In DSP, we represent an audio signal with ***floating-point*** (decimal) numbers and *scale* its values ***between -1.0 and +1.0***.

[↲] ***Mixing*** two signals can be done using the '**+**' operator.

[↲] When multiplying two signals together, we can already make a simple DSP effect: ***Amplitude Modulation*** (or ***Ring Modulation***).

[↲] Faust can process **multiple channels**, we can use the '**,**' symbol (***Parallel Operator***) to put signals *in parallel*.

[↲] We can declare **custom variables** (***functions***).

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

// We can declare our custom functions:
osc440 = os.oscrc(440) * 0.25;
osc880 = os.oscrc(880) * 0.25;

// And call them from anywhere:
process = osc440, osc880;
```

Normal

BASICS

[↲] In Faust, *connecting DSP functions* can be done using the ***sequential operator*** **':'**.

[↲] The ***split operator*** '<:' and ***cable operator*** can also be used to connect a signal to *multiple targets*.

[↲] On the other hand, the ***merge operator*** ':>' can be used to **merge** (**mix**) **signals together**.

[↲] *Graphical User Interface* (***GUI***) elements can be added to ***control parameters***: **sliders** (*hslider/vslider*), **buttons**, **switches...**

[↲] They can be used to make a proper ***gain*** control in ***dB*** for example.

[↲] ***Decibels*** (***dB***) are frequently used in audio to represent ***volume***, but they are sometimes a bit confusing to deal with.

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

process = os.sawtooth(440) : fi.lowpass(1, 440);
```

Normal

BASICS

[↲] In Faust, *connecting DSP functions* can be done using the ***sequential operator*** **':'**.

[↲] The ***split operator*** **'<:'** and ***cable operator*** can also be used to connect a signal to *multiple targets*.

[↲] On the other hand, the ***merge operator*** **':>'** can be used to **merge** (**mix**) **signals together.**

[↲] *Graphical User Interface* (***GUI***) elements can be added to ***control parameters***: **sliders** (*hslider/vslider*), **buttons**, **switches...**

[↲] They can be used to make a proper ***gain*** control in ***dB*** for example.

[↲] ***Decibels*** (***dB***) are frequently used in audio to represent ***volume***, but they are sometimes a bit confusing to deal with.

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

saw_filter = os.sawtooth(440) : fi.lowpass(1, 440);

process =  saw_filter <: _,_;
```

Normal

BASICS

[↲] In Faust, *connecting DSP functions* can be done using the ***sequential operator*** ':'.

[↲] The ***split operator*** '<:' and ***cable operator*** can also be used to connect a signal to *multiple targets*.

[↲] On the other hand, the ***merge operator*** ':>' can be used to **merge (mix) signals together.**

[↲] *Graphical User Interface* (***GUI***) elements can be added to ***control parameters***: **sliders** (*hslider/vslider*), **buttons**, **switches...**

[↲] They can be used to make a proper ***gain*** control in ***dB*** for example.

[↲] ***Decibels*** (***dB***) are frequently used in audio to represent ***volume***, but they are sometimes a bit confusing to deal with.

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

osc = (os.oscrc(440), os.oscrc(880)) :> _;

process =  osc * 0.25 <: _,_;
```

Normal

BASICS

[↲] In Faust, *connecting DSP functions* can be done using the ***sequential operator*** **':'**.

[↲] The ***split operator*** **'<:'** and ***cable operator*** can also be used to connect a signal to *multiple targets*.

[↲] On the other hand, the ***merge operator*** **':>'** can be used to **merge** (**mix**) **signals together.**

[↲] *Graphical User Interface* (***GUI***) elements can be added to ***control parameters***: **sliders** (*hslider/vslider*), **buttons**, **switches...**

[↲] They can be used to make a proper ***gain*** control in ***dB*** for example.

[↲] ***Decibels*** (***dB***) are frequently used in audio to represent ***volume***, but they are sometimes a bit confusing to deal with.

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

// hslider("name", default, min, max, step)
saw_freq = hslider("saw_freq", 440, 20, 10000, 1);
cutoff = hslider("cutoff", 440, 20, 10000, 1);

saw_filter = os.sawtooth(saw_freq) : fi.lowpass(1,
cutoff);

process =  saw_filter <: _,_;
```

Normal

BASICS

[↲] In Faust, *connecting DSP functions* can be done using the ***sequential operator*** ':'.

[↲] The ***split operator*** '<:' and ***cable operator*** can also be used to connect a signal to *multiple targets*.

[↲] On the other hand, the ***merge operator*** ':>' can be used to **merge (mix) signals together.**

[↲] *Graphical User Interface* (***GUI***) elements can be added to ***control parameters***: **sliders** (*hslider/vslider*), **buttons**, **switches...**

[↲] They can be used to make a proper ***gain*** control in ***dB*** for example.

[↲] ***Decibels*** (***dB***) are frequently used in audio to represent ***volume***, but they are sometimes a bit confusing to deal with.

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

// We add the '[style:knob]' metadata,
// and we also specify the unit:
gain = hslider("gain[style:knob][unit:dB]", -6, -96
, 6, 1) : ba.db2linear;

saw_freq = hslider("saw_freq[unit:Hz]", 440, 20, 10
000, 1);
cutoff = hslider("cutoff[unit:Hz]", 440, 20, 10000,
 1);

saw_filter = os.sawtooth(saw_freq) : fi.lowpass(1,
cutoff);

process =  saw_filter * gain <: _,_;
```

Normal

BASICS

[↲] In Faust, *connecting DSP functions* can be done using the ***sequential operator*** **':'**.

[↲] The ***split operator*** **'<:'** and ***cable operator*** can also be used to connect a signal to *multiple targets*.

[↲] On the other hand, the ***merge operator*** **':>'** can be used to **merge (mix) signals together.**

[↲] *Graphical User Interface* (***GUI***) elements can be added to ***control parameters***: **sliders** (*hslider/vslider*), **buttons**, **switches...**

[↲] They can be used to make a proper ***gain*** control in ***dB*** for example.

[↲] ***Decibels*** (***dB***) are frequently used in audio to represent ***volume***, but they are sometimes a bit confusing to deal with.

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

// We add the '[style:knob]' metadata,
// and we also specify the unit:
gain = hslider("gain[style:knob][unit:dB]", -6, -96
, 6, 1) : ba.db2linear;

saw_freq = hslider("saw_freq[unit:Hz]", 440, 20, 10
000, 1);
cutoff = hslider("cutoff[unit:Hz]", 440, 20, 10000,
 1);

saw_filter = os.sawtooth(saw_freq) : fi.lowpass(1,
cutoff);

process =  saw_filter * gain <: _,_;
```

Normal

SYNTHESIS

[↲] Faust has a in its libraries a good collection of '*basic*' **oscillators**, with different ***waveforms***: *sine*, *triangle*, *sawtooth*, *square*, *etc.*

[↲] When a Faust program starts to be a little more complex, its always good practice to ***refactor code*** by using ***custom functions*** with variable **parameters.**

[↲] In Faust, **functions** can take ***any element of the language as parameters***, including **GUI elements.**

[↲] Finally, the ***select*** primitive (an equivalent to ***switch*** in Max), allow to select an input from a list. It can be used in this case to *switch between waveforms*

[↲] Our goal now will be to ***apply this to our previous synthesizer.***

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

sine = os.osci(440) * 0.25;
triangle = os.triangle(440) * 0.25;
sawtooth = os.sawtooth(440) * 0.25;
square = os.square(440) * 0.25;

process =  sine;
// process = triangle;
// process = sawtooth;
// process = square;
```

Normal

SYNTHESIS

[↲] Faust has a in its libraries a good collection of '*basic*' **oscillators**, with different ***waveforms***: *sine*, *triangle*, *sawtooth*, *square*, *etc.*

[↲] When a Faust program starts to be a little more complex, its always good practice to ***refactor code*** by using ***custom functions*** with variable **parameters.**

[↲] In Faust, **functions** can take ***any element of the language as parameters***, including **GUI elements.**

[↲] Finally, the ***select*** primitive (an equivalent to ***switch*** in Max), allow to select an input from a list. It can be used in this case to *switch between waveforms*

[↲] Our goal now will be to ***apply this to our previous synthesizer.***

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

// Declaring functions, with variable parameters:
sine(freq) = os.osci(freq);
triangle(freq) = os.triangle(freq);
sawtooth(freq) = os.sawtooth(freq);
square(freq) = os.square(freq);

// Passing arguments from function to function:
synth(osc, freq, gain) = osc(freq) * gain;

// This way, we can modify everything here:
process =  synth(sine, 440, ba.db2linear(-16));
```

Normal

SYNTHESIS

[↲] Faust has a in its libraries a good collection of '*basic*' **oscillators**, with different ***waveforms***: *sine*, *triangle*, *sawtooth*, *square*, *etc.*

[↲] When a Faust program starts to be a little more complex, its always good practice to ***refactor code*** by using ***custom functions*** with variable **parameters.**

[↲] In Faust, **functions** can take ***any element of the language as parameters***, including **GUI elements.**

[↲] Finally, the ***select*** primitive (an equivalent to ***switch*** in Max), allow to select an input from a list. It can be used in this case to *switch between waveforms*

[↲] Our goal now will be to ***apply this to our previous synthesizer.***

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

sine(freq) = os.osci(freq);
triangle(freq) = os.triangle(freq);
sawtooth(freq) = os.sawtooth(freq);
square(freq) = os.square(freq);

synth(osc, freq, gain) = osc(freq) * gain : fi.lowp
ass(1, freq);

// Declare some controls:
freq_gui = nentry("frequency[unit:Hz]", 440, 20, 10
00, 1);
gain_gui = hslider("gain[style:knob][unit:dB]", -16
, -32, 0, 1) : ba.db2linear;

// We can pass GUI elements to a function:
process =  synth(square, freq_gui, gain_gui);
```

Normal

SYNTHESIS

[↲] Faust has a in its libraries a good collection of '*basic*' **oscillators**, with different ***waveforms***: *sine*, *triangle*, *sawtooth*, *square*, *etc.*

[↲] When a Faust program starts to be a little more complex, its always good practice to ***refactor code*** by using ***custom functions*** with variable **parameters.**

[↲] In Faust, **functions** can take ***any element of the language as parameters***, including **GUI elements.**

[↲] Finally, the ***select*** primitive (an equivalent to ***switch*** in Max), allow to select an input from a list. It can be used in this case to *switch between* ***waveforms***

[↲] Our goal now will be to ***apply this to our previous synthesizer.***

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

wave1 = os.triangle(440);
wave2 = os.square(440);

switch_gui = nentry("waveform", 0, 0, 1, 1);

process = wave1, wave2 : select2(switch_gui) * 0.25
;
```

Normal

SYNTHESIS

[↲] Faust has a in its libraries a good collection of '*basic*' **oscillators**, with different ***waveforms***: *sine*, *triangle*, *sawtooth*, *square*, *etc.*

[↲] When a Faust program starts to be a little more complex, its always good practice to ***refactor code*** by using ***custom functions*** with variable **parameters.**

[↲] In Faust, **functions** can take ***any element of the language as parameters***, including **GUI elements.**

[↲] Finally, the ***select*** primitive (an equivalent to ***switch*** in Max), allow to select an input from a list. It can be used in this case to *switch between* ***waveforms***

[↲] Our goal now will be to ***apply this to our previous synthesizer.***

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

sine(freq) = os.osci(freq);
triangle(freq) = os.triangle(freq);
sawtooth(freq) = os.sawtooth(freq);
square(freq) = os.square(freq);

oscillator(freq) = sine(freq), triangle(freq), sawt
ooth(freq), square(freq);

switch_gui  = nentry(
    "waveform[style:menu{'Sine':0; 'Triangle':1; 'S
awtooth':2; 'Square':3}]",
    0, 0, 3, 1
);

switch(freq) = oscillator(freq) : ba.selectn(4, swi
tch_gui);

freq_gui = nentry("frequency[unit:Hz]", 440, 20, 10
00, 1);
```

Normal

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

sine(freq) = os.osci(freq);
triangle(freq) = os.triangle(freq);
sawtooth(freq) = os.sawtooth(freq);
square(freq) = os.square(freq);

oscillator(freq) = sine(freq), triangle(freq), sawtooth(freq), square(freq);

switch_gui  = nentry(
    "waveform[style:menu{'Sine':0; 'Triangle':1; 'Sawtooth':2; 'Square':3}]",
    0, 0, 3, 1
);

switch(freq) = oscillator(freq) : ba.selectn(4, switch_gui);

freq_gui = nentry("frequency[unit:Hz]", 440, 20, 1000, 1);
gain_gui = hslider("gain[style:knob][unit:dB]", -16, -32, 0, 1) : ba.db2linear;

synth(freq, gain) = switch(freq) * gain : fi.lowpass(1, freq);

  Normal
```

FUNCTIONS

• ***Functions*** in programming languages usually consist in pieces of code containing a ***series of instructions to execute***. They can be called *once or multiple times*, *with or without variable parameters*, and can *return values*.

[↲] ***Function definitions*** in Faust have the syntax *function(parameter1, parameter2, ...) = expression;*".

[↲] In order to ***call*** (**execute**) that function, we need to ***replace its parameters (arguments)*** with the values we want to pass as parameters.

[↲] Arguments are only valid inside of the function definition, we say that they are *local to the **scope** of the function*. This also means that they ***take precedence*** over any other variable or expression that have the same name in the code.

FUNCTIONS

• ***Functions*** in programming languages usually consist in pieces of code containing a ***series of instructions to execute***. They can be called *once or multiple times*, *with or without variable parameters*, and can *return values*.

[↲] ***Function definitions*** in Faust have the syntax *function(parameter1, parameter2, ...) = expression;*".

[↲] In order to ***call*** (**execute**) that function, we need to ***replace its parameters (arguments)*** with the values we want to pass as parameters.

[↲] Arguments are only valid inside of the function definition, we say that they are *local to the **scope** of the function*. This also means that they ***take precedence*** over any other variable or expression that have the same name in the code.

editor control graph wave spectrum

faust

```
// Here, we define the function 'add',
// which takes 2 arguments (or parameters)
// that we call 'a' and 'b'.
// Note: 'a' and 'b' are arbitrary names,
// we could have used any name we wanted.
add(a, b) = a + b;
```

Normal

FUNCTIONS

• ***Functions*** in programming languages usually consist in pieces of code containing a ***series of instructions to execute***. They can be called *once or multiple times*, *with or without variable parameters*, and can *return values*.

[↲] ***Function definitions*** in Faust have the syntax *function(parameter1, parameter2, ...) = expression;*".

[↲] In order to ***call*** (**execute**) that function, we need to ***replace its parameters (arguments)*** with the values we want to pass as parameters.

[↲] Arguments are only valid inside of the function definition, we say that they are *local to the **scope** of the function*. This also means that they ***take precedence*** over any other variable or expression that have the same name in the code.

editor control graph wave spectrum

faust

```
add(a, b) = a + b;

// If we want to call (execute) that function,
// we need to replace 'a' & 'b' by the parameters
// we want to pass to the function.
// Here, we also store the result into a 'variable'
 called "result".
result = add(1, 2);

// Under the hood, 'a' and 'b' are replaced by '1'
and '2':
add(1, 2) = 1 + 2;
```

Normal

FUNCTIONS

• ***Functions*** in programming languages usually consist in pieces of code containing a ***series of instructions to execute***. They can be called *once or multiple times*, *with or without variable parameters*, and can *return values*.

[↲] ***Function definitions*** in Faust have the syntax *function(parameter1, parameter2, ...) = expression;*".

[↲] In order to ***call*** (**execute**) that function, we need to ***replace its parameters (arguments)*** with the values we want to pass as parameters.

[↲] Arguments are only valid inside of the function definition, we say that they are *local to the **scope** of the function*. This also means that they ***take precedence*** over any other variable or expression that have the same name in the code.

editor control graph wave spectrum

faust

```faust
a = 31;
b = 47;

add(a, b) = a + b;
result = add(1, 2); // result will be 3, not 78!

// On the other hand, if we were to change
// the name of the arguments of the 'add' function,
 like so:
add(c, d) = a + b;
// The result would be 78 this time,
// because 'a' and 'b' now refer to
// the 'a = 31' and 'b = 47' expressions.
```

Normal

FUNCTIONS

[↲] In Faust, everything is a ***function of time***. When a function is called in a Faust program, it will be called ***for each computation of a sample***, *e.g. 44100 times per second*.

[↲] In the end, all functions produces a numerical value (integer or floating point). Therefore, anything can be passed as a ***function argument***...

[↲] ... including functions!

editor control graph wave spectrum

faust

```
// Example of a constant function,
// it will produce the same output no matter
// how many times it is called:
my_constant_function = 440;

// We define a function 'oscillator'
// which itself calls the 'os.osci' function.
// os.osci computes for each new sample in time
// a new sinewave value.
oscillator = os.osci(440);

// Here, we sum the values of the two 'os.osci'
// functions, sample-by-sample:
mix = os.osci(440) + os.osci(880);
```

Normal

FUNCTIONS

[↲] In Faust, everything is a ***function of time***. When a function is called in a Faust program, it will be called ***for each computation of a sample***, *e.g. 44100 times per second*.

[↲] In the end, all functions produces a numerical value (integer or floating point). Therefore, anything can be passed as a ***function argument***...

[↲] ... including functions!

editor control graph wave spectrum

faust

```
// We can pass anything as a function argument in F
aust:
add(a, b) = a + b;

// We call 'add' on two sinewave oscillators
// with different frequencies.
// This will sum the two signals together:
process = add(os.osci(440), os.osci(880));

// It also works for GUI elements, like sliders:
slider_a = hslider("slider_a", 5, 0, 10, 1);
slider_b = hslider("slider_b", 5, 0, 5, 1);

process = add(slider_a, slider_b);
```

Normal

FUNCTIONS

[↲] In Faust, everything is a ***function of time***. When a function is called in a Faust program, it will be called ***for each computation of a sample***, *e.g. 44100 times per second*.

[↲] In the end, all functions produces a numerical value (integer or floating point). Therefore, anything can be passed as a ***function argument***...

[↲] ... including functions!

editor control graph wave spectrum

faust

```
// And... other functions!
// Here we declare the function 'osc_a',
// that takes on argument called 'freq',
// which we are going to pass through to the os.osci function.
// And yes 'os.osci' is also a function, which takes one argument (frequen
cy)
// and produces a sinewave at that frequency.
osc_a(freq) = os.osci(freq);

// Same here with a sawtooth, the name 'freq' doesn't conflict with the on
e
// from osc_a because they're only valid within their own scope:
osc_b(freq) = os.sawtooth(freq);

// We can call them like this:
process = osc_a(440) + osc_b(880);

// And what happens under the hood, if we unroll the function calls, will
be this:
osc_a(440) = os.osci(440);
osc_b(880) = os.sawtooth(880);

// We can also do the same thing in function calls:
add(a, b) = a + b;
osc_a(freq) = os.osci(freq);
osc_b(freq) = os.sawtooth(freq);

process = add(osc_a(440), osc_b(880));

// If we unroll the calls, we get this:
osc_a(440) = os.osci(440);
osc_b(880) = os.sawtooth(880);
add(osc_a(440), osc_b(880)) = osc_a(440) + osc_b(880);

  Normal
```

TIME

[↲] Faust has one final composition operator: the ***recursive operator*** ('~') which allows to ***connect two signals recursively*** (with ***a delay of one sample***).

[↲] The recursive operator is typically used to implement **counters**, and/or ***to represent time***. With it, we can count for instance *up to 44100 or 48000 samples* to ***represent one second of audio time***.

[↲] When implementing ***time-related functions***, it can prove very useful to ***monitor values from the GUI***.

[↲] We can for example use a counter to ***switch our synth's waveform automatically every second.***

editor control graph wave spectrum

faust

```
// We connect the signal '1' and an empty signal
// to the operator '+'.
// This expression has exactly one input
// and one output, same for an empty signal.
// Therefore, the two expressions can be
// mutually recursive:
process = (_ + 1) ~ _;
//        (  A  ) ~ B
```

Normal

TIME

[↲] Faust has one final composition operator: the ***recursive operator*** ('~') which allows to ***connect two signals recursively*** (with ***a delay of one sample***).

[↲] The recursive operator is typically used to implement **counters**, and/or ***to represent time***. With it, we can count for instance *up to 44100 or 48000 samples* to ***represent one second of audio time***.

[↲] When implementing ***time-related functions***, it can prove very useful to ***monitor values from the GUI***.

[↲] We can for example use a counter to ***switch our synth's waveform automatically every second***.

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

// We could use our previous definition
// of the counter, and wrap it around the
// value of the sample rate (using modulo)
// But there's going to be something wrong
// with it...
increment = (_ + 1) ~ _;
bad_counter = increment % 48000;

// This solution works:
counter = ((_ + 1) % 48000) ~ _;
//        (     A         ) ~ B

// We can put all of this in a function
// and make the limit of the counter variable:
counter(n) = ((_ + 1) % n) ~ _;

process = counter(ma.SR);


  Normal
```

TIME

[↲] Faust has one final composition operator: the ***recursive operator*** ('~') which allows to ***connect two signals recursively*** (with ***a delay of one sample***).

[↲] The recursive operator is typically used to implement **counters**, and/or ***to represent time***. With it, we can count for instance *up to 44100 or 48000 samples* to ***represent one second of audio time***.

[↲] When implementing ***time-related functions***, it can prove very useful to ***monitor values from the GUI***.

[↲] We can for example use a counter to ***switch our synth's waveform automatically every second.***

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

SR = 48000;
gui = vbargraph("value[style:numerical]", 0, SR);
counter(n) = ((_ + 1) % n) ~ _;

process = counter(SR): gui;
```

Normal

TIME

[↲] Faust has one final composition operator: the ***recursive operator*** ('~') which allows to ***connect two signals recursively*** (with ***a delay of one sample***).

[↲] The recursive operator is typically used to implement **counters**, and/or ***to represent time***. With it, we can count for instance *up to 44100 or 48000 samples* to ***represent one second of audio time***.

[↲] When implementing ***time-related functions***, it can prove very useful to ***monitor values from the GUI***.

[↲] We can for example use a counter to ***switch our synth's waveform automatically every second.***

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

SR = 48000;
gui = vbargraph("value[style:numerical]", 0, SR);

// We make the '1' (increment) variable:
counter(i, n) = ((_ + i) % n) ~ _;

// Outputs the value '1'
// whenever the counter reaches 48000
// (every second):
count1s = counter(1, SR) == 0;

process = counter(count1s, 4) : gui;
```

Normal

TIME

[↲] A counter can be used as a base for many many things. With it, we can for instance build a ***ramp signal*** that *goes from 0 to 1 repeatedly at a certain rate*.

• Notice anything? A *ramp* is very much like a *sawtooth oscillator*, only with a different range (0 to 1 instead of −1 to 1).

[↲] With a *ramp* (or *phasor*), we can pretty much already build ***all of the simple waveform oscillators***, with a few operations...

[↲] *Ramps* are often used as ***cursors*** to *read* or *write samples* in a ***buffer***. In Faust, *buffers* are implemented with the *rdtable* & *rwtable* primitives.

[↲] To playback a ***sound file***, we also typically use a phasor as a *read cursor*.

editor control graph wave spectrum

faust

```
counter(i, n) = ((_ + i) % n) ~ _;

// We can put a decimal as increment value,
// and wrap it around 0.0 and 1.0:
process = counter(0.00195, 1.0);

// But it would be much better to
// use the frequency unit in Hertz,
// We know one second of time is the sample rate,
// so to get the increment value, we can use:
frequency = 440/ma.SR;

process = counter(frequency, 1.0);
```

Normal

TIME

[↲] A counter can be used as a base for many many things. With it, we can for instance build a ***ramp signal*** that *goes from 0 to 1 repeatedly at a certain rate*.

• Notice anything? A *ramp* is very much like a *sawtooth oscillator*, only with a different range (0 to 1 instead of −1 to 1).

[↲] With a *ramp* (or *phasor*), we can pretty much already build ***all of the simple waveform oscillators***, with a few operations...

[↲] *Ramps* are often used as ***cursors*** to *read* or *write samples* in a ***buffer***. In Faust, *buffers* are implemented with the *rdtable* & *rwtable* primitives.

[↲] To playback a ***sound file***, we also typically use a phasor as a *read cursor*.

editor control graph wave spectrum

faust

```
counter(i, n) = ((_ + i) % n) ~ _;

// We can put a decimal as increment value,
// and wrap it around 0.0 and 1.0:
process = counter(0.00195, 1.0);

// But it would be much better to
// use the frequency unit in Hertz,
// We know one second of time is the sample rate,
// so to get the increment value, we can use:
frequency = 440/ma.SR;

process = counter(frequency, 1.0);
```

Normal

TIME

[↲] A counter can be used as a base for many many things. With it, we can for instance build a ***ramp signal*** that *goes from 0 to 1 repeatedly at a certain rate*.

• Notice anything? A *ramp* is very much like a *sawtooth oscillator*, only with a different range (0 to 1 instead of −1 to 1).

[↲] With a *ramp* (or *phasor*), we can pretty much already build ***all of the simple waveform oscillators***, with a few operations...

[↲] *Ramps* are often used as ***cursors*** to *read* or *write samples* in a ***buffer***. In Faust, *buffers* are implemented with the *rdtable* & *rwtable* primitives.

[↲] To playback a ***sound file***, we also typically use a phasor as a *read cursor*.

editor control graph wave spectrum

faust

```
import("stdfaust.lib");

counter(i, n) = ((_ + i) % n) ~ _;
phasor(f) = counter(f/ma.SR, 1.0);

// For sawtooth, we re-scale the signal to [-1, 1]
sawtooth(f) = phasor(f) * 2.0 - 1.0;

// Sine function has an input range
// of [0, 2*PI]:
sine(f) = sin(phasor(f) * 2.0 * ma.PI);

// For square, we can use a simple 'if' expression
// output -1.0 when phase is below 0.5
// and 1.0 when it's above:
square(f) = ba.if(
    phasor(f) > 0.5, // condition
    1.0, // then
   -1.0  // else
);
```

Normal