EPRO/IAS

Bachelor Studiengang Medientechnik
Fachhochschule St.Pölten

Patrik Lechner

Wien, December 5, 2017

1. Introduction

Figure 1.1.: Weird effects of digital signals in video

This section tries to make sure we are all on the same page. It goes through some mathematics notation and principles of digital audio.
If you rather want a video format revision of digital audio (I mean, it's 2017, of course you want), I can recommend D/A and A/D | Digital Show and Tell (Monty Montgomery @ xiph.org) [1]. Some of the points in the video go beyond what we are doing here and vice versa, so don't purely rely on the video though.
Ideally, if you throughly understood the introduction section, you should be able to go through the remaining chapters with tempo.

1.1. About this document

Please report any mistakes, errors etc to ptrk.lechner@gmail.com.

Some information in this document is relevant for understanding its contents but not relevant for the exam. For example in chapter 2 we will find a really complicated result of an equation. The point there is just that it's complicated. And it is shown how complicated, but this is nothing to be learned by heart. Importance is tried to be made clear using the following formats:

Text, figures, and equations with a gray background like this are background information that is not to be learned by heart.

Video Analogies
This document tries to explain digital signals. It does this by use of audio signals mainly. Sometimes video analogies are given. These are also not relevant for the exam.

Very important things are framed.

pd objects will always be in courier new and square brackets. Like this: `[metro]`

1.1.1. Pure data Version

This document was written using some version of *pd-extended*. It is now in the process of migrating everything to version:
Pd 0.47.1, everything tested on a Linux Debian system. The following Libraries will be used:

[1] https://www.youtube.com/watch?v=cIQ9IXSUzuM

- cyclone
- zexy

1.2. About plotting signals

We will need to plot a lot of signals in order to understand them better. Most of the time, such a plot will look like figure 1.2.

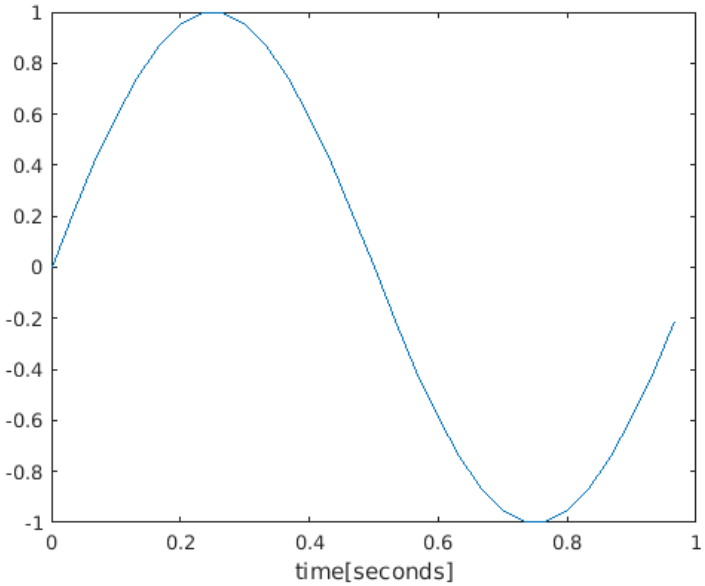


Figure 1.2.: Sine wave, 1Hz, sampled at 30Hz sample rate

This plot looks nice but it has a problem. The sine wave is sampled at a sampling rate of 30 Hz, but we see a continuous line. This "connection of the dots" is created by plotting. It is kind of similar to what our *digital-to-analog-converter* does. It somehow[2] interpolates the values we have.
This can be misleading, so we should actually plot something more like figure 1.3. Often we can see plots that look like figure 1.4 as well when a signal is analyzed.

[2] linear interpolation in case of the plot

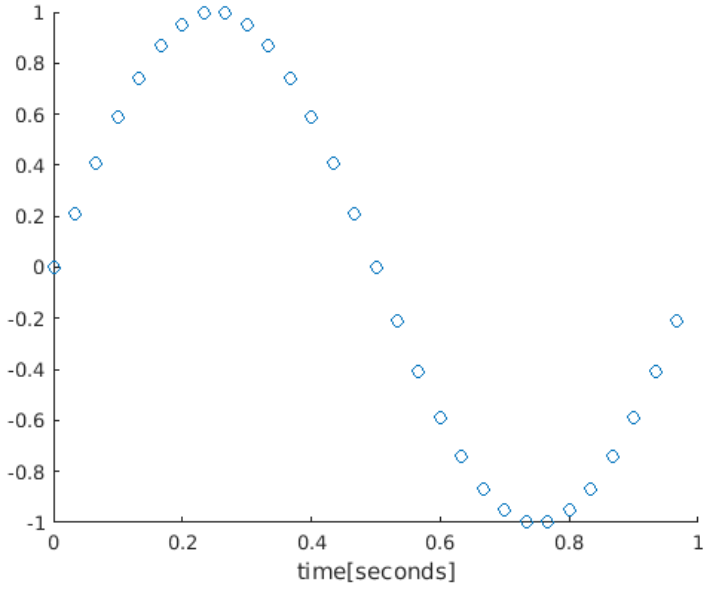


Figure 1.3.: Sine wave, 1Hz, sample rate 30Hz, scatter-plot

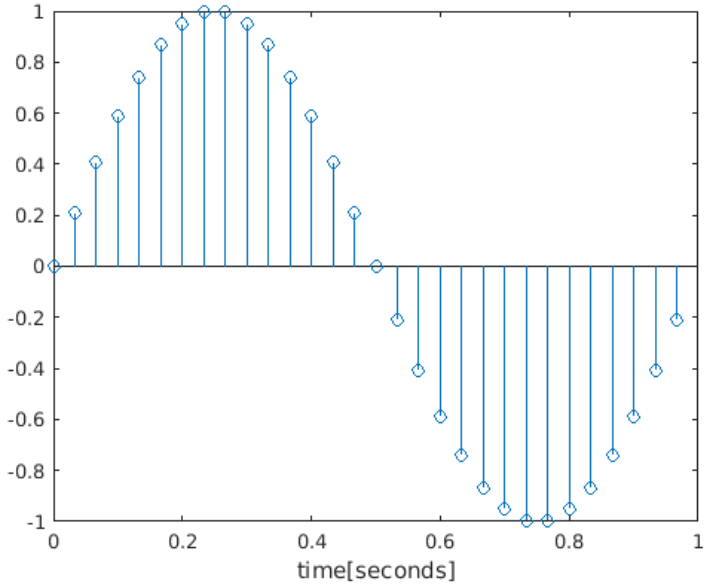


Figure 1.4.: Sine wave, 1Hz, sample rate 30Hz, stem-plot

So why don't we always do a stem- or scatter-plot? Simply because it gets too crowded with our usual sampling rates in audio. It just works with very low sampling rates or very short signals.

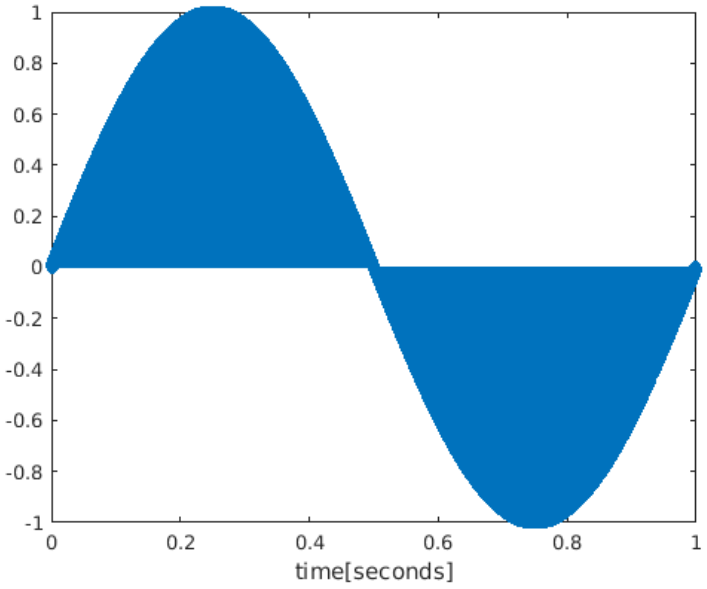


Figure 1.5.: Sine wave, 1Hz, sample rate 44100Hz, stem-plot

But we should never forget that we don't actually have the values in between the dots. Digital signals are not defined between their sampled points.

1.3. What is aliasing?

Aliasing in audio means problems caused by signals that exceed the nyquist rate.
The nyquist rate,let's call it f_n for now, is defined by the half of the sample rate (f_s).
So,

$$f_n = \frac{f_s}{2} \tag{1.1}$$

A digital system can only describe signals up to his nyquist rate. If we try to make signals higher than this frequency, we will fail and encounter strange effects.
Visually speaking, frequencies higher than nyquist fold back. So, let's assume we have a sampling rate of 100Hz. Nyquist would be at 50Hz. If we try to synthesize a sine wave with 51Hz, what we will get is a 49Hz one. If we try to make a 52Hz one, we will get 48Hz. So you see, it simply folds back.

Video analogies
Aliasing in graphics usually means *spacial* aliasing, so aliasing in the space domain. This is what we see in figure 1.6. But there is also time domain aliasing in film. It is actually more natural to think about the sampling rate in audio as the same as the frame rate in video. For some really weird effects that arise in video due to time domain aliasing see airplane[3], Water experiment[4]or Guitar strings[5].

Figure 1.6.: Spacial aliasing in graphics. The "frequency" of the intended pixels is to high for the actual pixels.

The particularly strange effects in the airplane example above are caused by the rolling shutter of a CMOS sensor. Since it is not sampling the incoming light uniformly (at the same time) the image is distorted.

In figure 1.7 you can see a visualization of aliasing in the time domain.

[2]https://www.youtube.com/watch?v=LVwmtwZLG88
[3]https://www.youtube.com/watch?v=GBtHeR-hY9Y
[4]https://www.youtube.com/watch?v=jcOKTTnOIV8

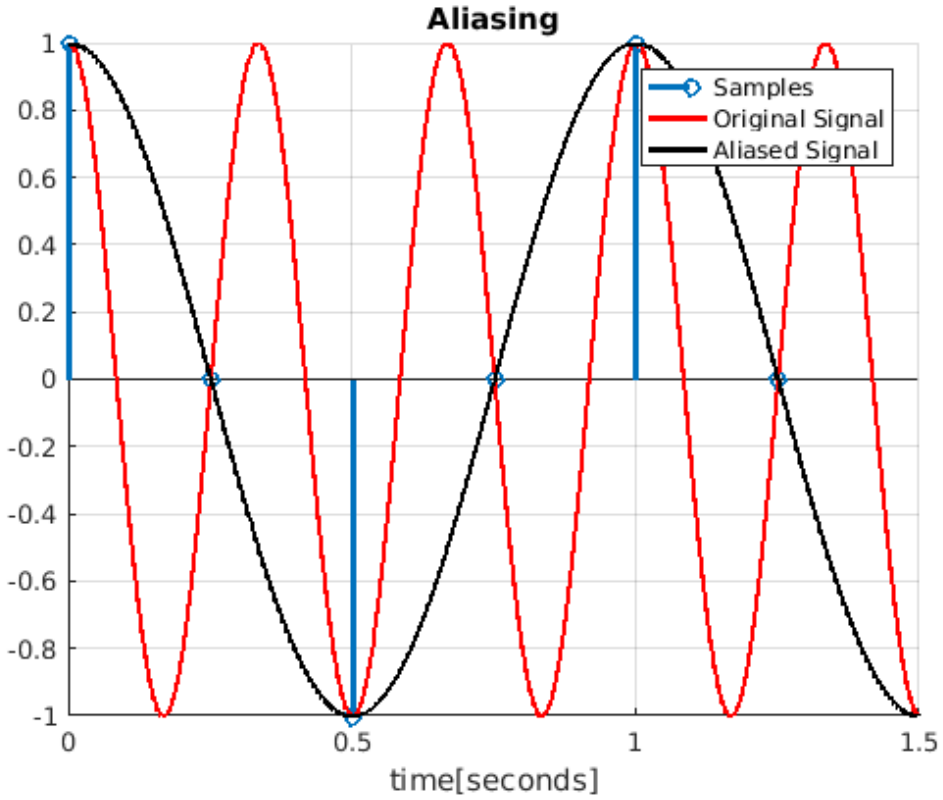


Figure 1.7.: Aliasing visualized in the time domain. A sampling rate of 4 Hz is used. Therefore frequencies will fold above 2 Hz, which is the nyquist frequency. The input cosine has a frequency of 3 Hz, labeled “Orginal Signal”. What we would get is the sampled points. If these are digital-to-analog converted, we would get what is labeled “Aliased signal ” in this plot, so a 1 Hz cosine.

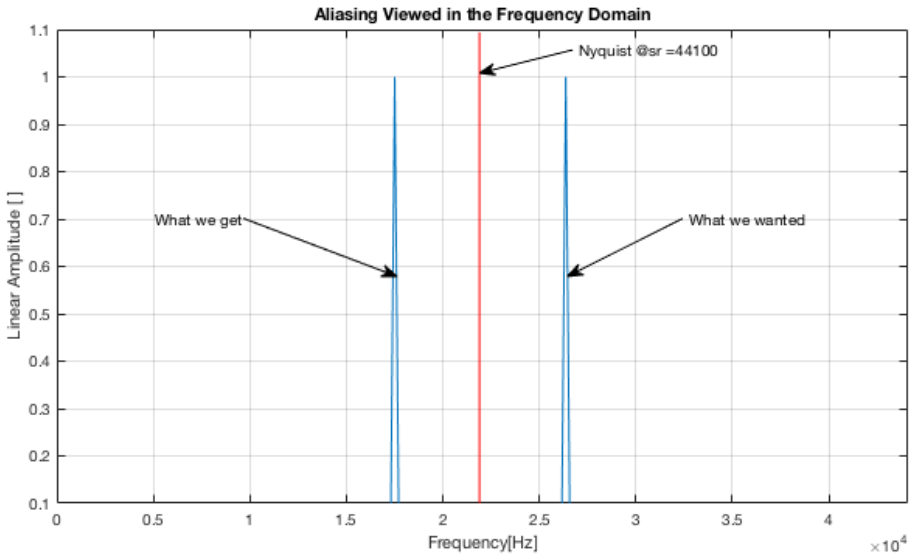


Figure 1.8.: Here we can see the phenomenon of aliasing in the frequency domain. We try to synthesize a sine wave at about 26kHz. At a sampling rate of 44100 Hz, this is not possible, nyquist is at 22050 Hz. What we actually get is a sine wave at about 17kHz. In this plot, the nyquist frequency is marked with the red line. Please note that the distance between the red line to both peaks is identical. This is why the phenomeno is also called *fold-back*

1.4. Scaling and Mapping Signals

It is an important skill to be able to scale signals from one range to another. We need it a lot and we will be able to think about signals more easily if we mastered this task. It's actually quite simple, we just have to imagine the signals visually.
So what exactly do we have to do here? We are confronted with the following problem: Given some signal, say, a sine wave with its maximum at the value 1 and its minimum at the value -1. How to bring it to a different range, say, 0-10?
It helps me a lot to solve this problem in two parts: first get the input in the range 0-1, then from there go to the desired range. What can we do to the signal? Let's take a sine wave:
Well we can add and subtract to move the wave vertically, so let's add 1 to move it up: So we can move signals around by adding constant values. We can scale them by multiplication. So if we take our sine that now ranges from 0 to 2 and multiply it by 0.5, we get whats in figure 1.11.
Using what we got now in figure 1.11, we can just multiply by 10, easy! Beware that there are always multiple solutions to this kind of a problem. Try to find another one

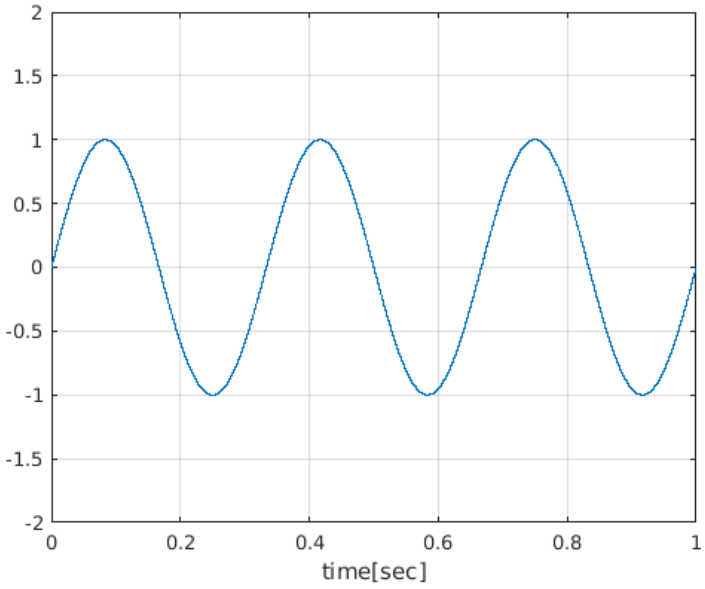


Figure 1.9.: a sine wave

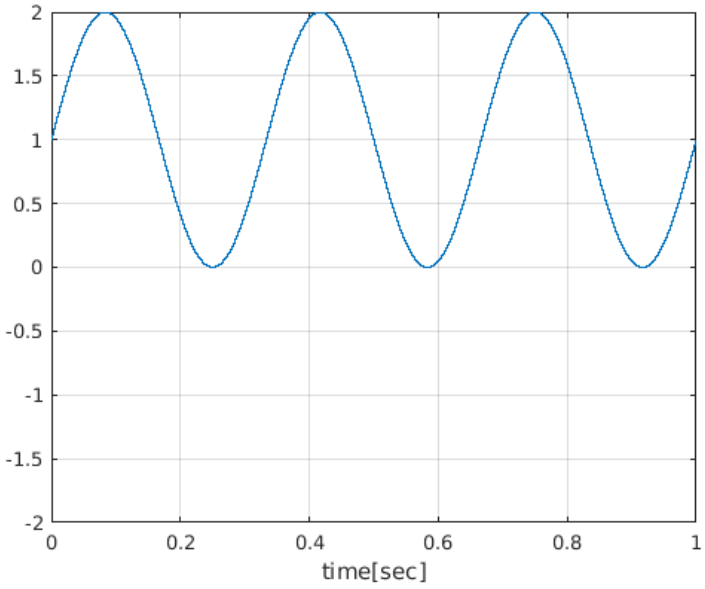


Figure 1.10.: the same sine wave, 1 added to a each sample, therefore shifted upwards.

for the problem above!

Question 1 *Let's take a sine wave that has it's minimum at 2 and it's maximum at 5. What do we have to do to get it into a -1 to 1 range?*

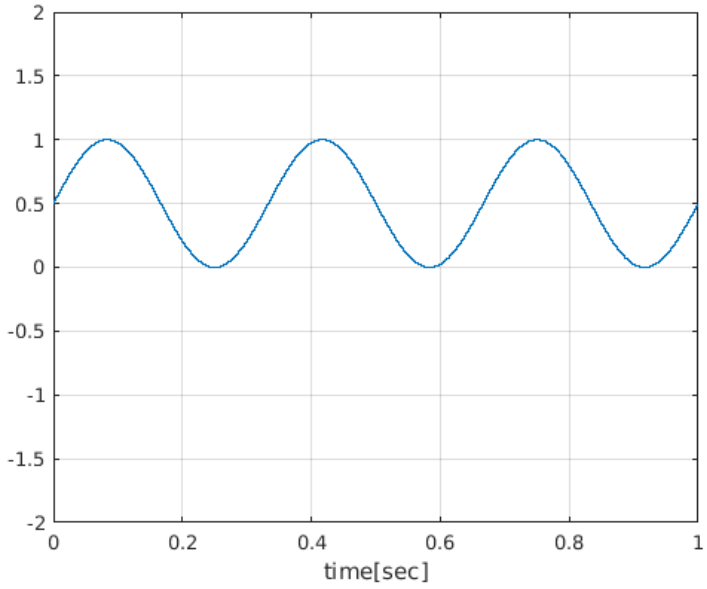


Figure 1.11.: Sine with a range of 0-1. Obtained by taking a sine wave, adding 1 and dividing by two afterwards.

Answer 1 *We could subtract 3.5 to center the wave around zero first. Afterwards we take care of the amplitude by multiplying by $\frac{2}{3}$ (since the initial wave has a peak-to-peak amplitude of three and we want a peak-to-peak amplitude of 2)*

1.5. What's DC-Offset?

What we did above by adding a constant value to a signal can be called adding DC-offset ("Gleichspannungsversatz"), DC-Bias or a DC component. These are different words for the same thing.
DC-offset can also be encountered in signals we recorded (caused by old or broken equipment mainly). But we have seen that we can also generate DC-offset.
To state it again clearly:

> DC-Offset is a constant value over time, or a constant offset from the zero value in Y. So for example in figure 1.11 we see a DC-offset of 0.5 since subtracting this value would center the wave around 0.

If we try to think how this kind of signal looks in the frequency domain we find that it is energy at 0Hz. In figure 1.12 you can see a couple of cosine waves plotted. This should hep imagine that a constant signal, a signal that does not move a all, can be described by a cosine with 0 Hz and a certain amplitude.

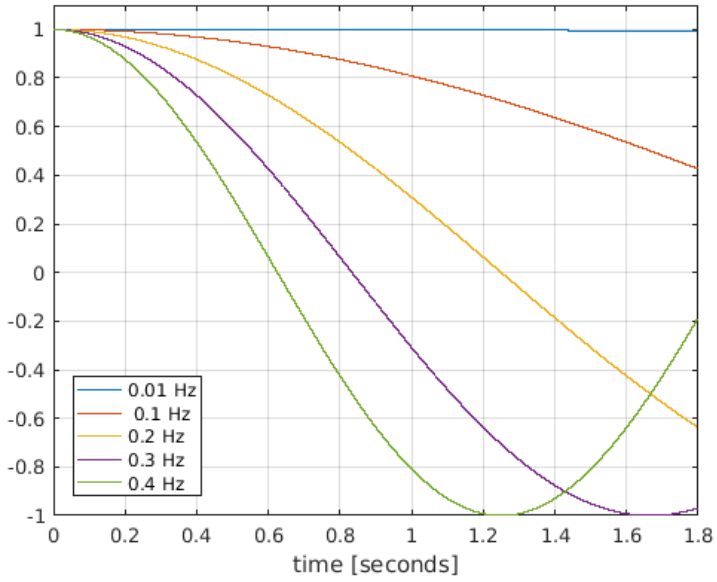


Figure 1.12.: Cosines with different very low frequencies approaching 0Hz, so DC offset.

Let's quickly state this differently, so we can appreciate the surprising connections between DC-offset and an impulse. A DC-offset signal is a signal consisting of constant values, so for example $\{..., 1, 1, 1, 1, 1, 1, ...\}$. Its spectrum is a single peak at 0Hz, so the spectrum's values look like $\{..., 0, 0, 0, 1, 0, 0, 0, ...\}$[6]

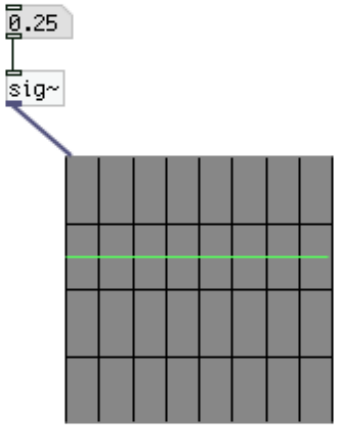


Figure 1.13.: One of the ways to generate a DC signal in pd.

[6]Why is the 1, so the 0 Hz component not at the beginning of the list? Or is it not 0 Hz? The 1 is supposed to be at 0Hz, and it is centered in this list because doing an FFT actually always gets us a two-sided spectrum. You can ignore this fact until the FFT chapter, and pretend that the list starts with 1 (which wouldn't be wrong also).

1.6. What's an Impulse?

Impulses are very useful signals. We can produce an impulse using the `[dirac~]` object.

Figure 1.14.: The `[dirac~]` object produces an impulse if we send it a `bang`.

Different people will define what's an impulse in different ways:

- A sound engineer might tell you, that clapping your hands or shooting with a gun creates an impulse
- a mathematician will maybe give you the definition of the *dirac delta function* (that's where the name for pd's impulse object comes from)
- Somebody working with discrete (so digital) signals will give you rather the definition of the Kroneker delta function.

For us, strictly speaking, an impulse will be a signal that contains only zeros, but one sample with the value one. We can see such an impulse in figure 1.15. On the x-axis, we have the time in samples[7].

[7] Don't be irritated by the fact that the sample numbers on the x-axis are ranging from -5 to 5. You can just as well imagine them going from 0 to 10 or 1 to 11. It doesn't really matter. However, this way of displaying an impulse is very common since if the impulse is filtered, it's symmetry is an important factor.

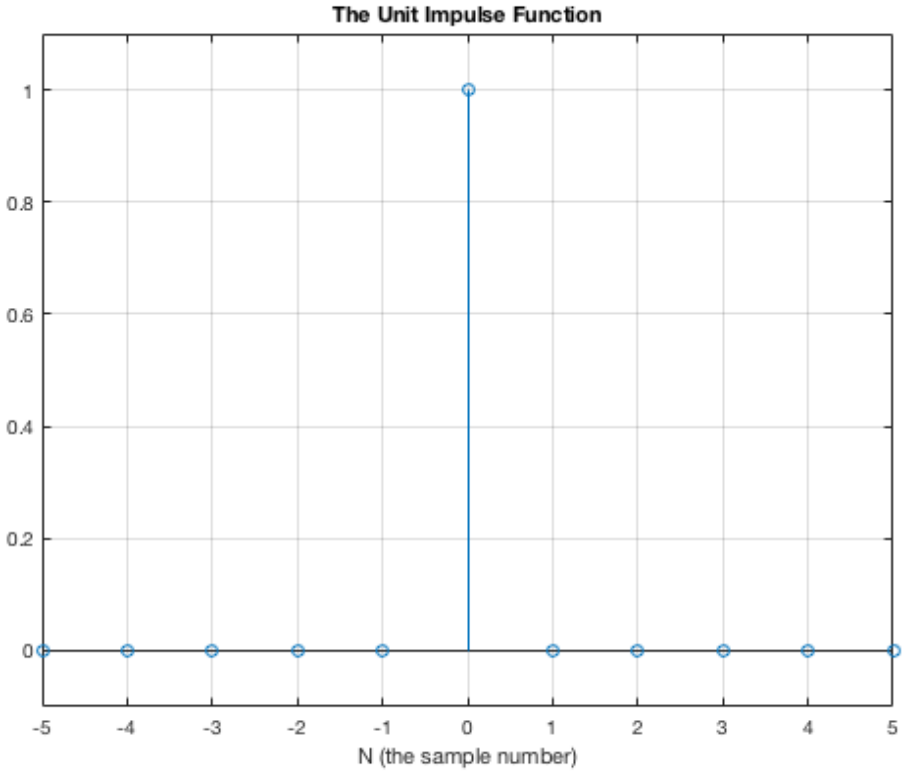


Figure 1.15.: caption

So in the time domain, an impulse's samples have the values $\{..., 0, 0, 0, 1, 0, 0, 0, ...\}$. If we look at it's spectrum, we will find that it contains all frequencies, there is just a flat line in the spectrum (you can see an impulse's spectrum in figure 3.13 also.). The spectrum has a *constant value*, something like $\{.., 1, 1, 1, 1, 1, 1, ...\}$. Do these two lists look familiar? Right, it is exactly the opposite of the DC-offset signal.

1.7. How to describe audio mathematically

If we want to talk about signals, or if we want to analyze them, it is often useful to look at the problem mathematically. First, let's introduce some conventions. They might look unfamiliar or complicated. But in fact it is not very complicated and knowing these conventions makes it easy to communicate (e.g. reading scientific papers about our topics or explaining something to another person).

1.7.1. signals

Usually we describe a *digital* signal by a name, say x, (but you can call it however you want). If we want to talk about the individual samples, or values of the (mono) signal, we can use a subscript or parenthesis. So if the fifth sample of x is 1, we could write:

$$x_5 = 1 \tag{1.2}$$

or

$$x(5) = 1 \tag{1.3}$$

Oftentimes we like to talk about a signal more generally and we use n as a place holder for this index, so we might write x_n, meaning the nth sample of x.

sine waves

We use sine waves a lot. The syntax can get a bit overwhelming at first, so let's quickly explain what's going on in a standard sine wave oscillator.

$$x(t) = A \cdot sin(2\pi ft + \phi) \tag{1.4}$$

Where f is the frequency in Hertz and t is the time in seconds. ϕ Is a (possibly constant) phase offset and A can be used to scale the whole thing. Since cosine and sine have their peaks at -1 and 1, A will be the amplitude. If A is set to 0.5, the resulting signal will have it's peaks at -0.5 and 0.5.
The upper equation is rather complete. Most of the time, we will just ignor the phase and the amplitude and just write:

$$x(t) = sin(2\pi ft) \tag{1.5}$$

Sometimes, we will simplify even more and just write $sin(a)$ or $sin(b)$ or similar.

In literature, we sometimes encounter $x(t) = sin(t \cdot \omega)$. ω stands for a "frequency"(actually for the rotational speed) in *radians per second*. 2π radians per second are 1 Hz and we can therefore convert radians per second to frequency, v, with:

$$v = \omega/2\pi \tag{1.6}$$

This notation is only covered since it is very common, but it will not be used here, since it is considered more intuitive to work with frequency in Hz.

What is actually the significance of using either *sin* or *cos*?
In most cases for us, this does not matter at all. The difference between sine and

cosine is just in phase. Since we are most of the time describing a sine (or cosine) wave oscillator as a function of time, the only difference between the two would be their phase if given some moment in time. This is nothing one could hear also.

1.7.2. systems

There are many ways to describe systems. Digital LTI systems (Linear, time invariant Systems) can be described using

- difference equations
- Block diagrams
- Transfer Functions
- and other things.

For us, the most important ways to describe a system are block diagrams and difference equations.

1.7.3. systems

There are multiple ways in which we can describe linear time invariant (LTI) systems. We will talk more about this in chapter 3. Here, we will just look at how *difference equations* work. These are one possibility to describe a digital LTI system. They are the discrete equivalent of difference equations. It's really not that complicated:

$$y(n) = x(n) \tag{1.7}$$

Would be an equation that describes a system that does nothing. it takes the input sample, $x(n)$ does nothing and defines the output sample $y(n)$ with it. Another really simple system would multiply its input by two:

$$y(n) = x(n) \cdot 2 \tag{1.8}$$

It's getting interesting if we start playing with the index:

$$y(n) = x(n-1) \tag{1.9}$$

Is a delay by one sample.

$$y(n) = x(n) + x(n-1) \tag{1.10}$$

Takes it's input sample and the previous input sample, adds them together and spits it out. Can you imagine what this does? We'll get to it in chapter 3...

This is just a preview, but it really gets interesting if we start working with feedback:

$$y(n) = x(n) + y(n-1) \tag{1.11}$$

1.8. Message Domain/Signal Domain

This is not pure data specific although it might sound like it. In pure data we can have audio signals. Audio signals are processed in buffers. They are just numbers, but these numbers are calculated at a rate of 44100 Hz *all the time* if we choose to have our sample rate at 44100 Hz.
Messages on the other hand are not calculated that often and not all the time. Messages are processed *on demand*. They are event based. This means that, for example, if we hit a note on our midi keyboard or enter a number in a `[Numberbox]` this *event* will pass through its following objects *once*. Have a look at figure 1.16 or at the patcher. Also note that pd helps us in understanding if we deal with message or signal domain:

- pd is indicating the signal domain with thicker patch cords
- signal domain inlets/outlets are black (look closely)
- objects that deal with the signal domain have a tilde (~) at the end of their name.

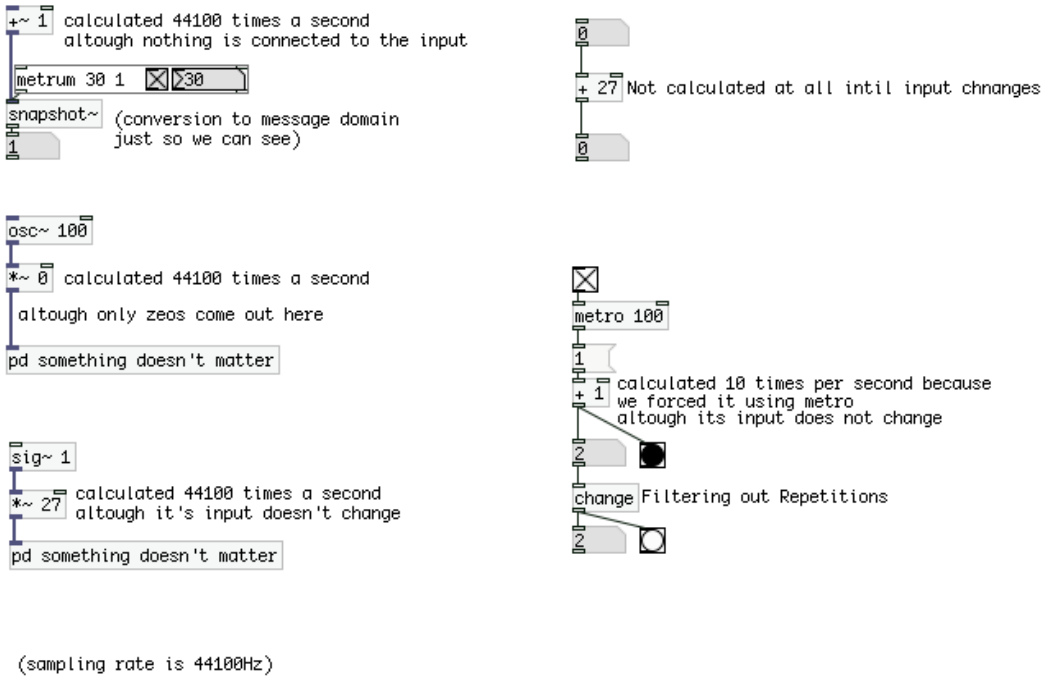


Figure 1.16.: The patcher `messageDomainSignalDomain.pd` should demonstrate the differences between message domain and signal domain in pure data.

How is this not specific to pure data? The key points here are not specific to this programming language. If we, for example adjust the volume of a track in pro tools or ableton live or similar, then the information of our fader movement will also be in some kind of message domain.
One key aspect here is: some kind of conversion between the two is necessary if we want to switch between them. If we want to control the volume of an audio signal from the message domain we have a problem: we will get noisy output since the message domain is not running at sample rate. Look at a naive attempt to controlling the amplitude of a sine wave:

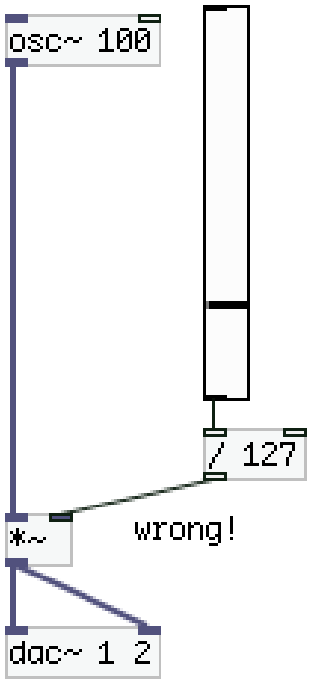


Figure 1.17.: The patcher `amplitudeWrongRight.pd`

And let's look at what kind of waveform it will produce:

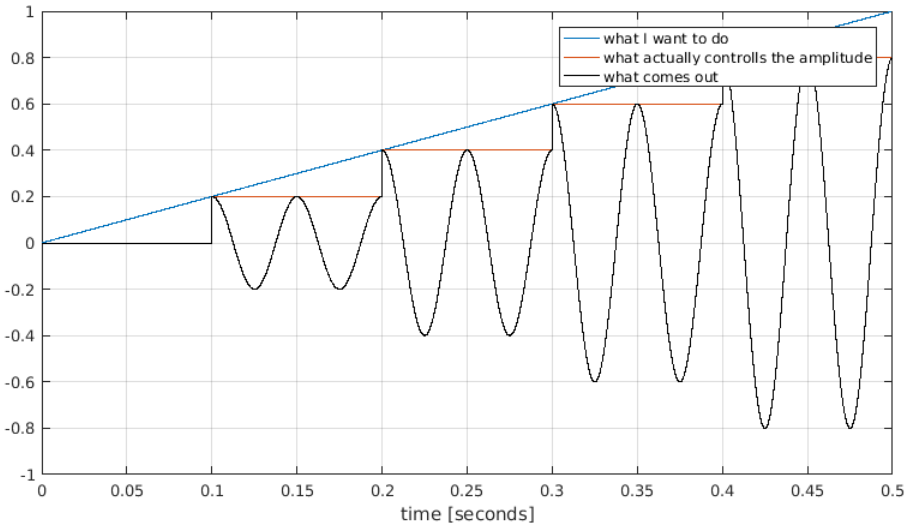


Figure 1.18.: This is a cosine wave at 20 Hz and the message domain running at 100 millisecond interval to make the effect more extreme and visible

This is a problem, since the sudden jumps produce clicks, crackling or zipper noise (reproduce or open the patcher and hear it for yourself!). These clicks are caused by the fact that the message domain just doesn't produce the numbers at the rate of the sampling rate. It's too slow, so to speak.
In pure data (but also in general) this problem can be solved by interpolation. pure data offers the `[line~]` object for this situation. It can be used to create signals at sampling rate and we can tell it to ramp to a specified value in a given amount of time. In figure 1.19 we see it ramping to any value that comes from the `[/ 127]` object within 20 milliseconds[8]. This interpolation reduces the clicks and noise dramatically. Note that the output of `[line~]` is in the signal domain (thick cable).

[8] 20ms is just an example. It is a reasonable time for this kind of interpolation but really, it is just an example. It could just as well be 50 ms.

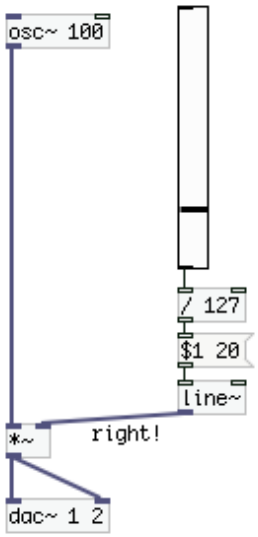


Figure 1.19.: Controlling the amplitude of an oscillator. The interpolation done with `[line~]` suppresses clicks.

1.9. Key Points

- Make sure you understand and know the mathematical notation. It will be how we talk in the remaining chapters.
- Make sure you understand what aliasing means in audio.
- Make sure you are comfortable with basic mathematical operations on signals, such as adding constants and multiplying with constants
- Make sure you understand the difference between message and signal domain and the problem with using a message domain "signal" controlling an audio stream.
- Make sure you know what the spectrum and time domain signal of an impulse and DC-Offset look like.

Contents

1. Introduction **II**

1.1. About this document III
1.1.1. Pure data Version III
1.2. About plotting signals IV
1.3. What is aliasing? VI
1.4. Scaling and Mapping Signals IX
1.5. What's DC-Offset? XI
1.6. What's an Impulse? XIII
1.7. How to describe audio mathematically XIV
1.7.1. signals XV
1.7.2. systems XVI
1.7.3. systems XVI
1.8. Message Domain/Signal Domain XVII
1.9. Key Points XX

I. Semester 3

1. Sampling, waveshaping, and non-linearity **2**

1.1. Waveshaping 2
1.1.1. The simplest case: a linear Transfer function. 3
1.1.2. Simple non-linearity: X^2 7
1.1.3. How can waveshaping be implemented? 9
1.1.4. How is Waveshaping related to other techniques? 10
1.1.5. Why is Waveshaping useful? 12
1.1.6. What are the problems with waveshaping? 12
1.2. Sampler 14
1.2.1. Granular Sampling 18
1.3. Key Points 19
1.4. Hausübung 20
1.4.1. Testmodul 20

2. Modulation **21**

2.1. AM 22

2.2. FM 26
2.3. Key Points 32

3. Filters **33**
3.1. Seeing Frequency Content in the Time Domain 34
3.2. Ways of Describing a Filter 36
3.2.1. Difference Equations 37
3.2.2. Block Diagrams 38
3.2.3. pd Code 41
3.2.4. Impulse Response 41
3.2.5. Text Oriented Code 41
3.3. Ways of Getting an Intuitive Understanding 42
3.3.1. Combfilter to Lowpass 42
3.3.2. Prolonging an Impulse 45
3.3.3. A moving Average 46
3.3.4. A sine at Nyquist 46
3.3.5. Highpass 47
3.3.6. Video Filters 48
3.4. Filters in pd 53
3.4.1. Lowpass 53
3.4.2. Highpass 53
3.4.3. Bandpass 54
3.5. Types of Digital Filters 54
3.6. Key Points 56

4. Fourier Transformation **57**
4.1. Notizen 57
4.2. Fourier Transformation 58

Part I.

Semester 3

1. Sampling, waveshaping, and non-linearity

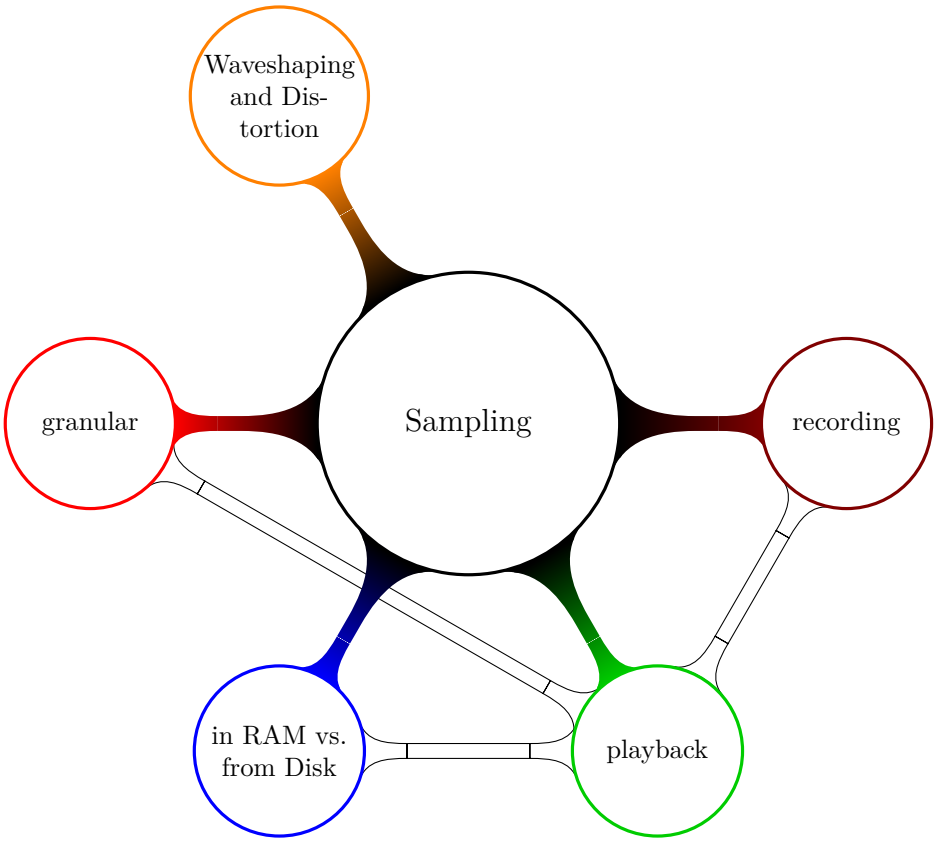


Figure 1.1.: Lecture Contents

1.1. Waveshaping

Wikipedia quote, page “waveshaper”:
„The mathematics of non-linear operations on audio signals is difficult, and not well understood.“ Waveshaping means distortion. It adds overtones, take a look at figure 1.2.

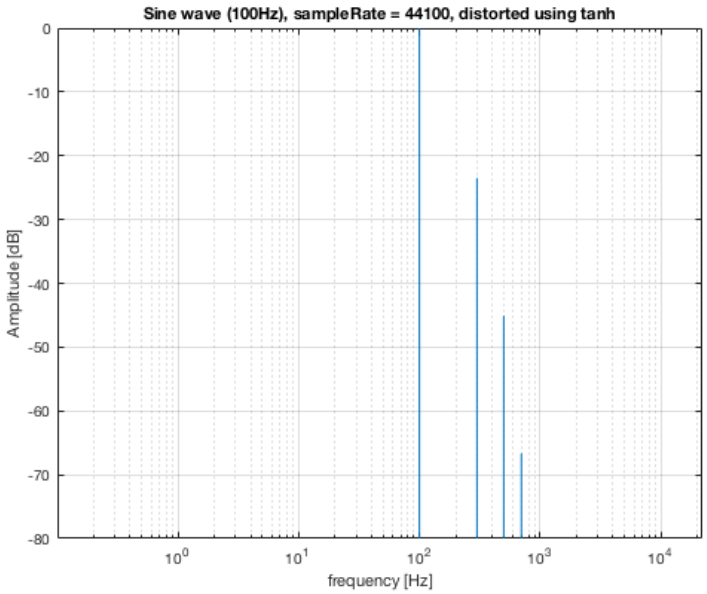


Figure 1.2.: A sine wave has been generated and waveshaping was applied to add overtones.

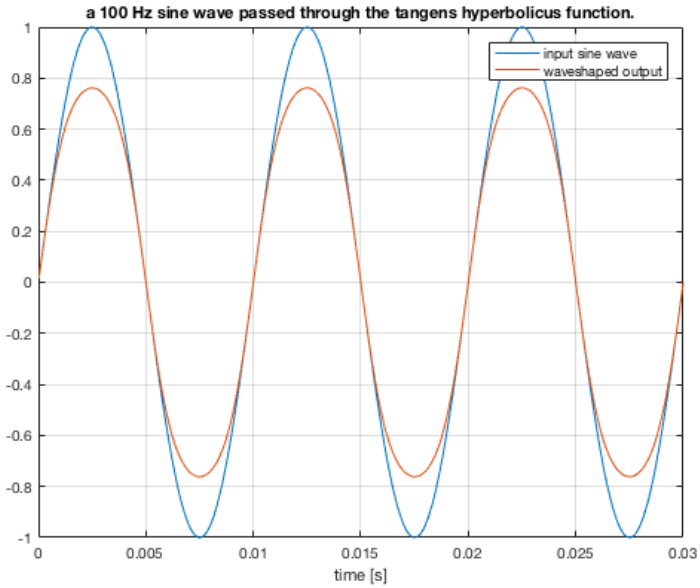


Figure 1.3.: The same as the spectrogram above, but in the time domain. We can see the input sine wave and the slightly distorted output. It may look like just the amplitude has changed, but the sine's actual *shape* has changed slightly

1.1.1. The simplest case: a linear Transfer function.

See 1.4. A linear transfer function is used as a lookup table for a sinosoidal input.

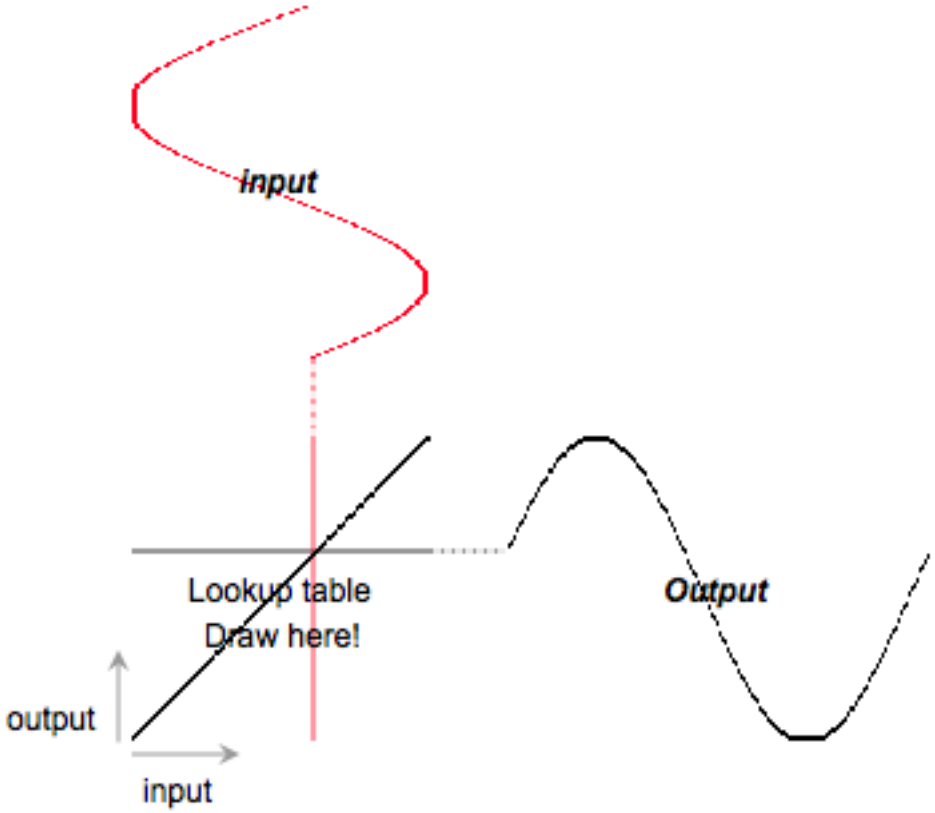


Figure 1.4.: Linear Transfer function

A transfer function in the sense of a waveshaper (a "transfer function" might also mean frequency response in other contexts) is a simple look-up function. Waveshaping means to use an input wave to *look up* values in a table or function. A linear transfer function, let's call it l, can result in no change, for example, it might return $l(x) = x$. This means, that whatever value we pass in, we get the same value out. Other linear transfer functions might *only* change the amplitude. For example $f(x) = x \cdot \frac{1}{2}$. That doesn't seem very interesting. But it might explain the term "linear". A transfer function is linear if it looks like a line if we plot it. Look at figure 1.5.
Non-liner transfer functions behave differently. They map their input to other values, such as $f(x) = x^2$. And if we plot then, they don't look like a line. You can also look at figure 1.6 in order to understand what's happening. We again see a linear transfer function but also a non linear one.

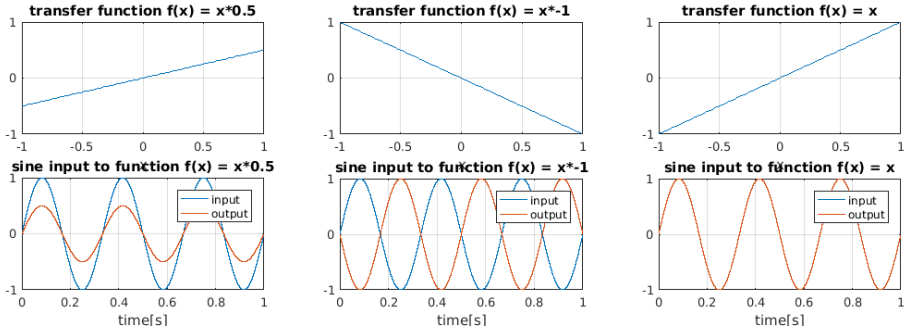


Figure 1.5.: A couple of linear transfer functions and their corresponding effects demonstrated using a sine wave. From left to right: multiplication by 0.5, so attenuation by about 6dB, inversion, and the "do-nothing"-function.

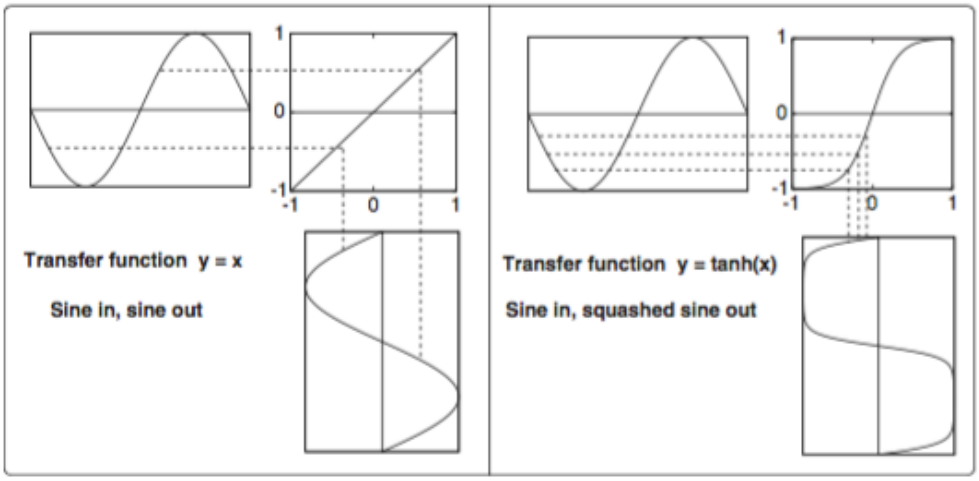


Figure 1.6.: A waveshaping visualization taken from Farnell (2010)

But let's get back to our square function, since it's simpler and we will find some surprising results when analyzing it. Let's first simply plot it too.

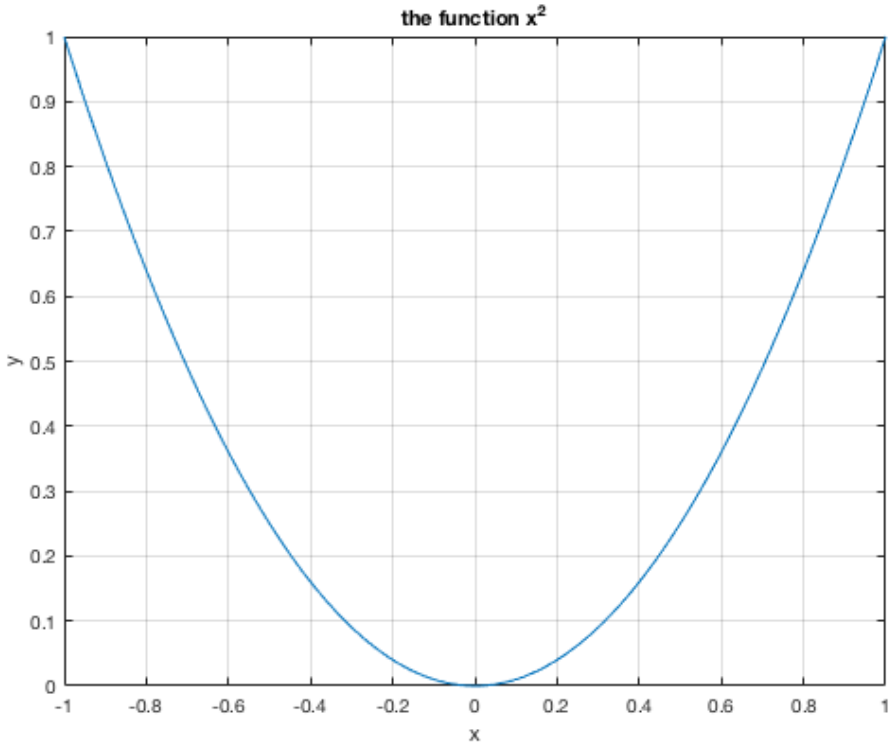


Figure 1.7.: The function $f(x) = x^2$

Video Analogies

What does applying a curve to every pixel of a video or image do? Is this even something people do? Of course! Think of contrast curves, below is an example from photoshop. Beware that video is working with unipolar input values and audio with bipolar inputs.

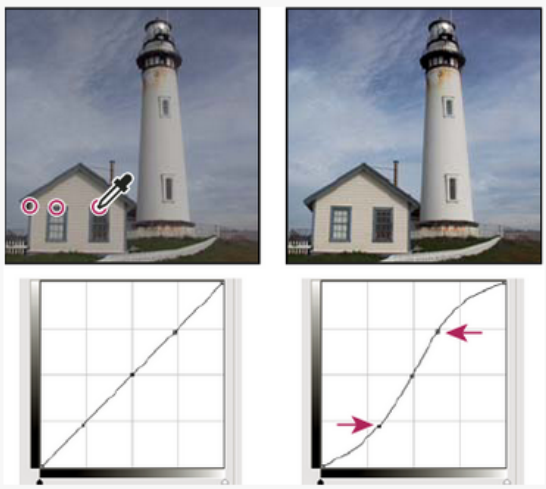


Figure 1.8.: Contrast curve in photoshop

1.1.2. Simple non-linearity: X^2

So let's analyze what happens if we use this function for waveshaping. Here it is again:

$$f(x) = x^2 \tag{1.1}$$

Let's simply listen to what's happening, building it in pd:

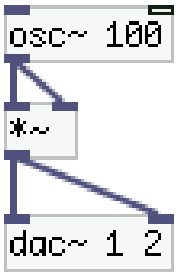


Figure 1.9.: The square function in pure data, using a 100Hz sine as a test signal. What do you hear?

And we can simply plot what happened if we apply the function before also trying to understand analytically:

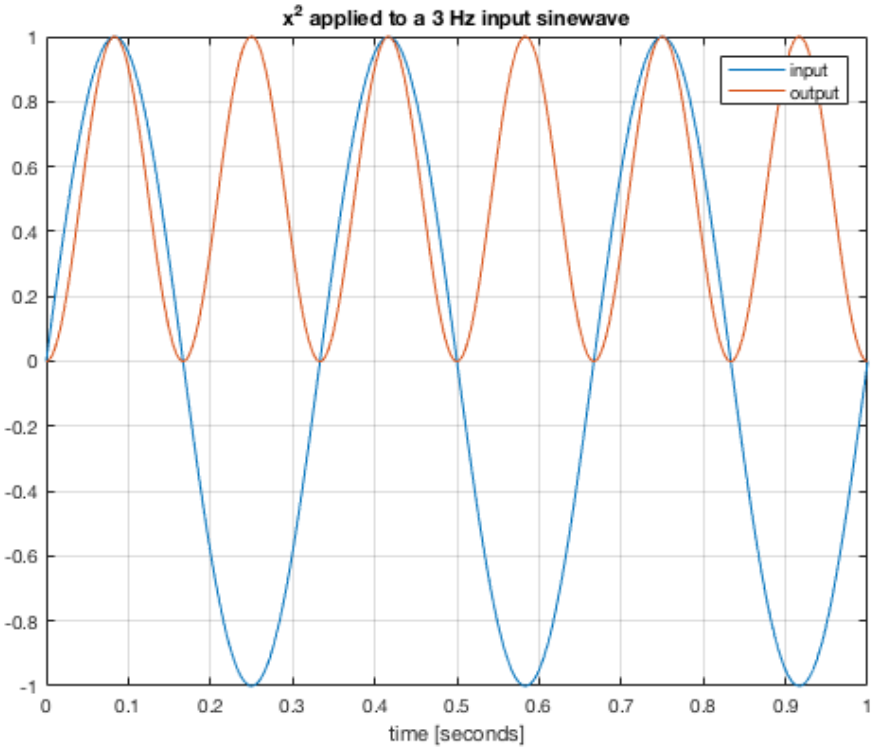


Figure 1.10.: Applying the square function to an input sine wave.

Weird, the input seems to double in frequency (did you hear that?). Let's try to understand what's happening.
So we calculate what happens if we send a cosine through this function, so let's take:

$$x = cos(\omega) \tag{1.2}$$

wit arbitrary ω. We can just ignore ω here for a while. Usually, there should be some indexing variable in the cosine function if we want to describe an oscillator that moves over time, but let's also skip that.
So applying our square function we of course get:

$$y = cos(\omega)^2 \tag{1.3}$$

This again results in:

$$y = cos(\omega) \cdot cos(\omega) \tag{1.4}$$

So far so trivial. Note that a multiplication of two oscillators is called *Amplitude Modulation* (actually, in this case we encounter "Ring Modulation", but let's ignore that also), and we know things about Amplitude modulation, namely:

When multiplying two oscillators, we get sum and difference of the two input frequencies. (And the whole output is attenuated by 6dB)

The above statement in equation form:

$$cos(a) \cdot cos(b) = \frac{cos(a+b) + cos(a-b)}{2} \tag{1.5}$$

We could also have looked up this *trigonometric identity*. This means for our experiment with our cosine squared:

$$y = \frac{cos(\omega + \omega) + cos(\omega - \omega)}{2} \tag{1.6}$$

So:

$$y = \frac{cos(2 \cdot \omega) + cos(0)}{2} = \frac{cos(2 \cdot \omega)}{2} + \frac{1}{2} \tag{1.7}$$

We arrive at the same result! But is this true for every input? That would mean we just built a frequency shifter, did we? No. Waveshaping is much more complicated.

This is immediately obvious when we try to do the same with two oscillators:

$$x = cos(\omega_1) + cos(\omega_2) \tag{1.8}$$

then

$$y = (cos(\omega_1) + cos(\omega_2))^2 \tag{1.9}$$

$$y = cos(\omega_1)^2 + cos(\omega_2)^2 + 2 \cdot cos(\omega_1) \cdot cos(\omega_2) \tag{1.10}$$

And finally:

$$y = \frac{cos(2 \cdot \omega_1)}{2} + \frac{1}{2} + \frac{cos(2 \cdot \omega_2)}{2} + \frac{1}{2} + 2 \cdot (\frac{cos(\omega_1 + \omega_2) + cos(\omega_1 - \omega_2)}{2}) \tag{1.11}$$

1.1.3. How can waveshaping be implemented?

Take a look at figure 1.11. What do you think is happening? On the left side, we see waveshaping as we did it above, using a mathematical function, in this case the tangens hyperbolicus, to distort our signal. On the right side, we see a table that contains the authors desperate attempt to draw the same function with the mouse. The results are theoretically equivalent (if the function in the table was correct), but what are the advantages and disadvantages of the two approaches? Also, be sure to understand what the addition of 1 and the multiplication with 50 does on the right side. Hint: the array has 100 points.

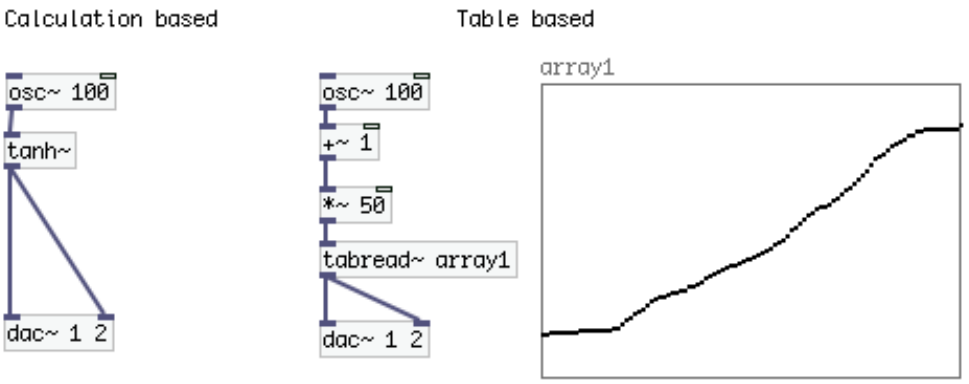


Figure 1.11.: Left: using a mathematical function to calculate the output. Right: using a table to look up the output.

1.1.4. How is Waveshaping related to other techniques?

Sampling

If we take a look at figure 1.19, we see that we play a sound file by accessing a buffer (wavetable) using an index, an oscillator. This is effectively the same setup as we would build for distorting an input sound. Also take a look at figure 1.12, which showing us that waveshaping and wavetable synthesis are identical.

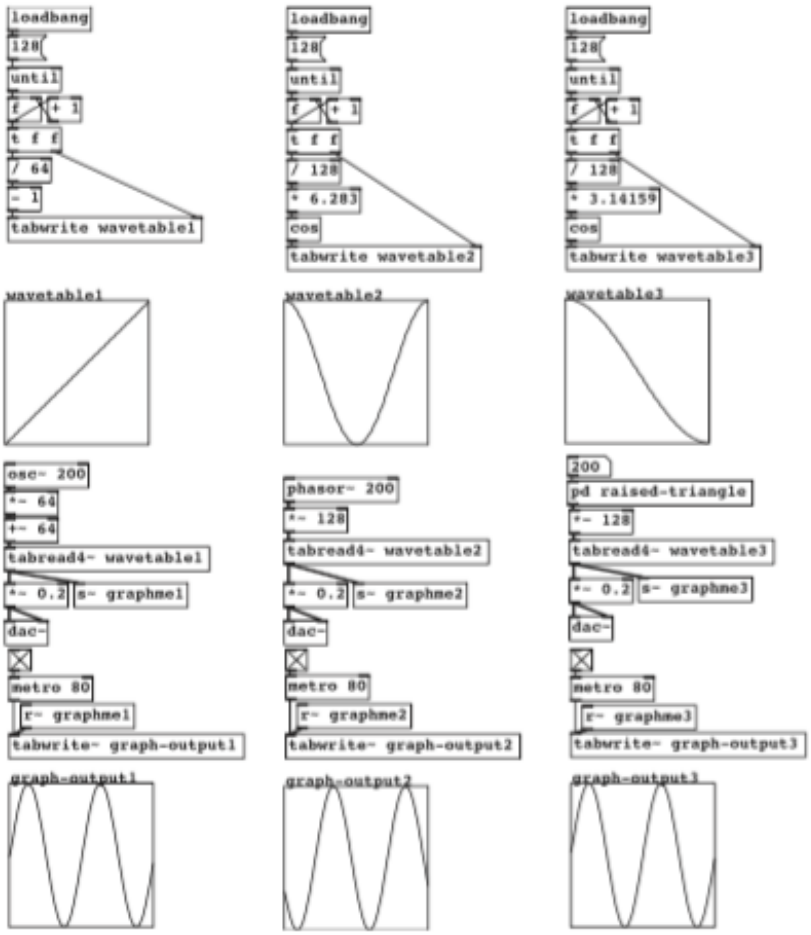


Figure 1.12.: Picture taken from Farnell (2010), showing the identity of waveshaping and wavetable synthesis

Modulation

While we will talk about modulation in a separate chapter, let's loosely define amplitude modulation (AM) as the multiplication of two oscillators and frequency modulation (FM) as varying the frequency of an oscillator using another oscillator. So, as we have also seen above, AM looks like this:

$$y = sin(a) \cdot sin(b) \tag{1.12}$$

and FM looks like this:

$$y = sin(sin(a)) \tag{1.13}$$

in practice, the a and b terms are a bit more complicated, but we will look at this later. That certain cases of AM are identical to waveshaping has been shown above, think

about the square function again. This of course does not mean that waveshaping can do everything AM can do and it does not mean that AM can achieve everything that waveshaping can. This should just show that we can understand the techniques from the perspective of another. What about FM? Well if our lookup function we use for waveshaping is a sine wave, we arrive at the exact same equation as how we defined FM above. Again, practically speaking, the results we get with these two techniques are very different, but we can see the connections.

1.1.5. Why is Waveshaping useful?

The output spectrum is dependent on the input amplitude. This makes it easy to create complex evolving spectra.

1.1.6. What are the problems with waveshaping?

Waveshaping adds overtones. When we build a waveshaper, we have to be aware of aliasing. Take a look at figures 1.13 and 1.14. Sinewaves have been amplified and clipped here.

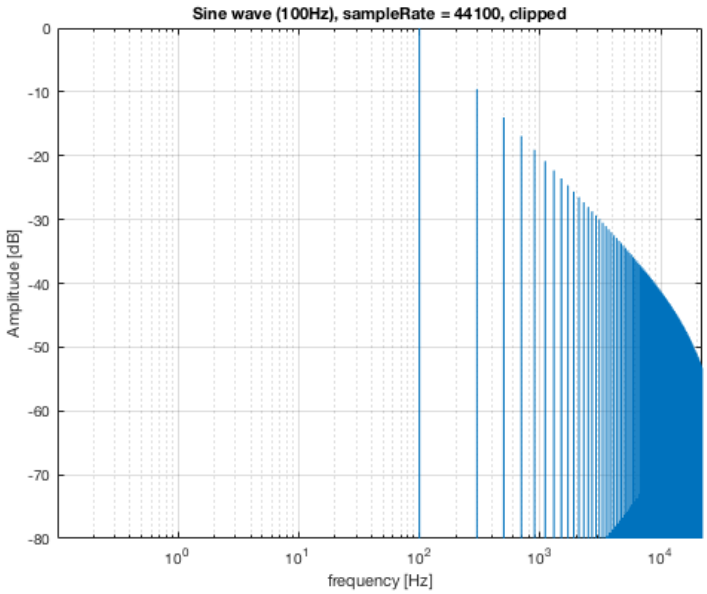


Figure 1.13.: A sine wave was generated and clipped. Clipping is a form of waveshaping which adds many overtones. Note how high frequencies fold back into the lower parts of the spectrum because they exceed the Nyquist-rate.

In pd we could achieve this like in figure 1.15.

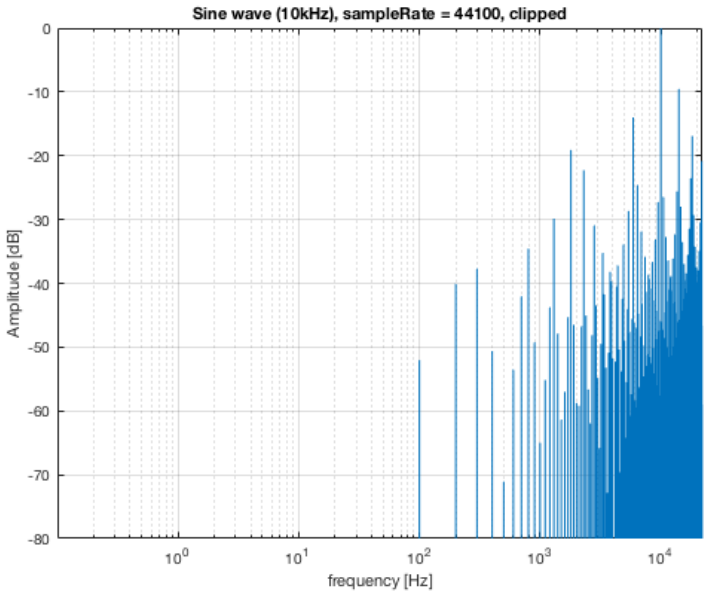


Figure 1.14.: Again, a sine wave, this time with a higher frequency to begin with. Extreme clipping has been applied by boosting the input amplitude. The aliased overtones are all over the place, even below the input frequency.

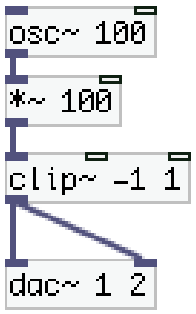


Figure 1.15.: Clipping an amplified sine wave in pd

What does the output look like? Let's not only look at the spectra but also at the time signal:

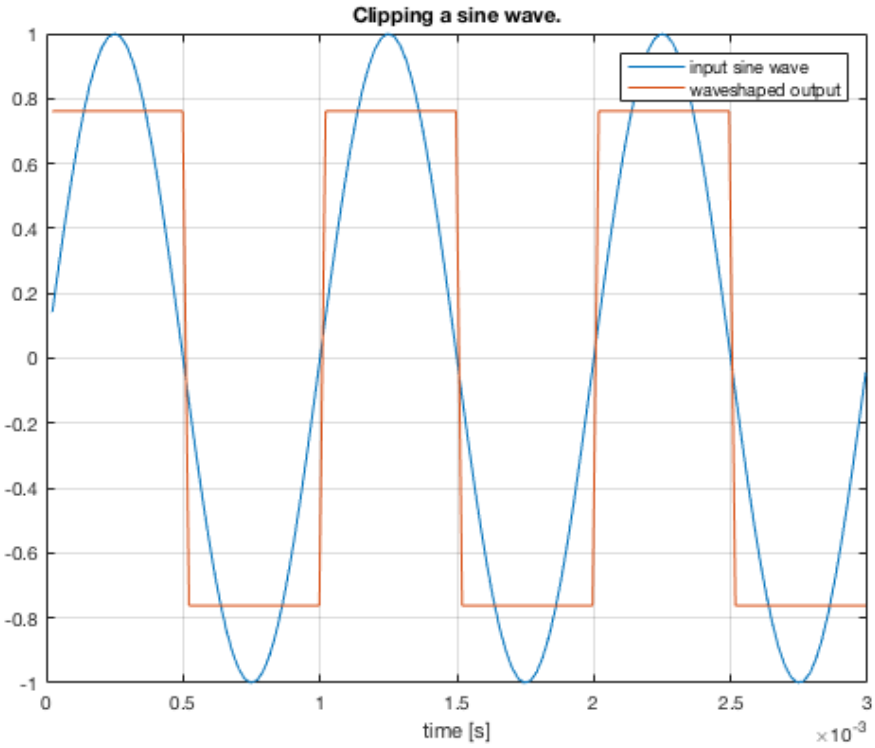


Figure 1.16.: An amplified and clipped sine wave in the time domain.

We see that we can arrive at a square-wave like result, but this square-wave is not anti-aliased.

The problem of aliasing in waveshaping is usually treated by over-sampling. This does not solve the problem but lessens it significantly resulting in cleaner, arguably better sound. Oversampling means that, if we work at a sample-rate of 44.1kHz, the input is up-sampled, essentially interpolated, to be at a sample-rate of 88.2kHz. Then the waveshaping is applied, leaving room for high frequencies up to 44.1kHz. Using a lowpass filter, high frequencies over 22.1kHz are then attenuated as much as possible, in order to be able to down-sample again to reach our initial sample-rate of 44.1kHz. To state it more simple: Waveshaping is usually encapsulated in a process that runs at higher sampling rates in order to lessen aliasing.

1.2. Sampler

In pure data (but also in general) we can choose to play a sound from RAM or more or less[1] directly from the hard disk. Both Approaches have their different use-cases.

[1] the data played from disk needs to be cached also

Video Analogies

In video, we need to make the same decision essentially. We can choose to buffer as much as we can on the VRAM of the graphics card or rather play from disk. Also different codecs specialize on balancing the load differently. The HAP codec for example tries to only push as much of the load to the hard disk and as little as possible to the CPU (which is decoding the video for playback). In times of SSDs this is a very good (and scalable) method to push a lot of data to playback in realtime.

Work in progress.

Figure 1.17.: simpleSampler

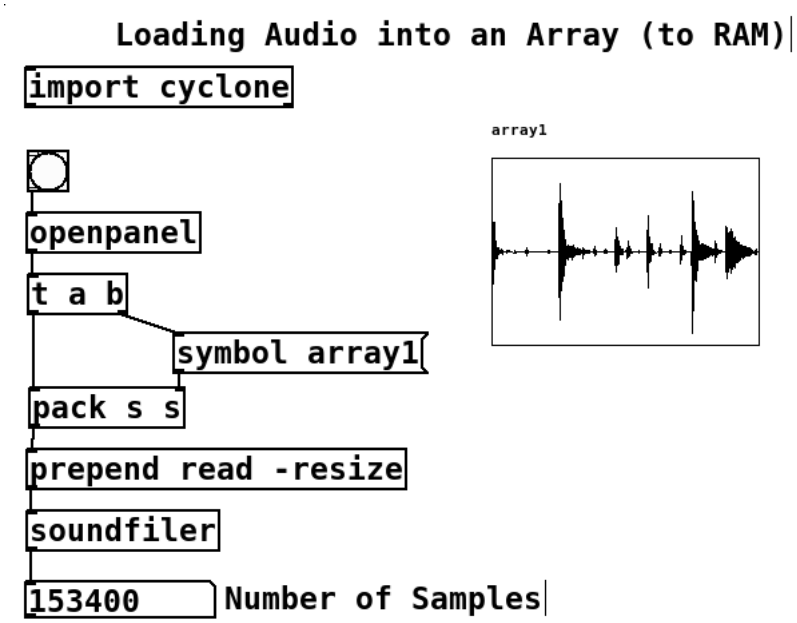


Figure 1.18.: sound in Ram

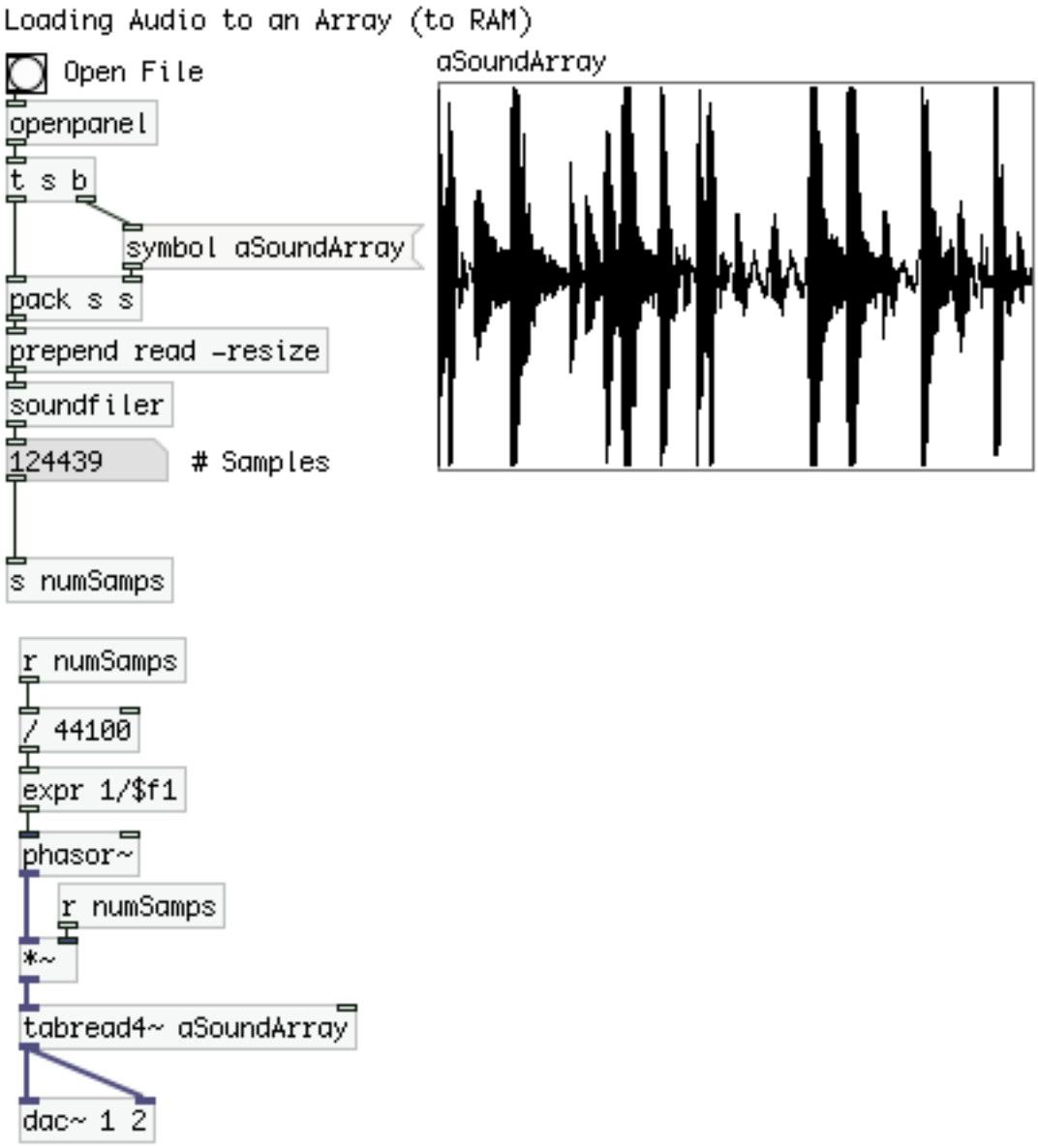


Figure 1.19.: RamFilePlayback

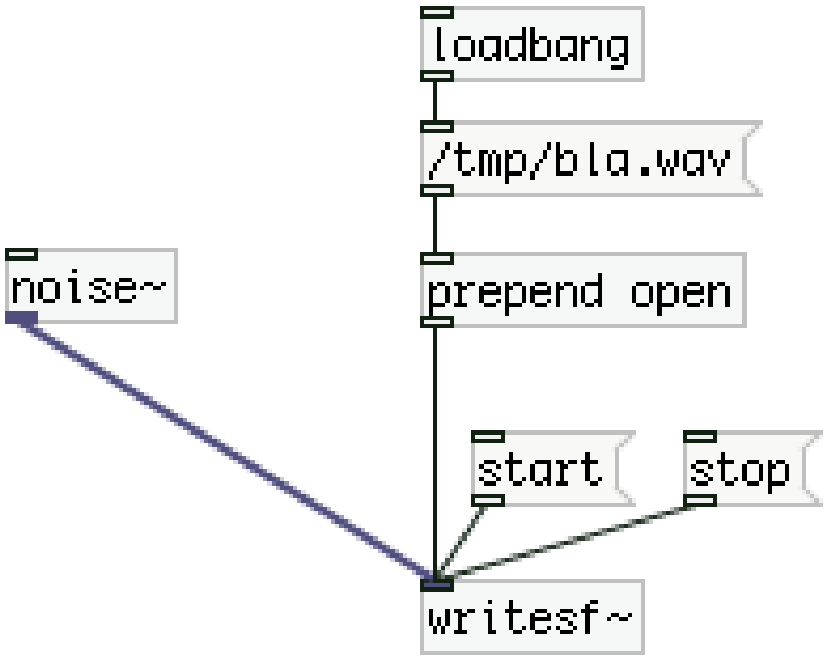


Figure 1.20.: writing Audio to disk

1.2.1. Granular Sampling

In figure 1.21 you see the basic structure of granular sampling. Granular sampling allows us for example to playback a file slower without lowering the pitch. It is one of the methods to make speed and ppitch independent from each other during playback. Other methods also include FFT based approaches.

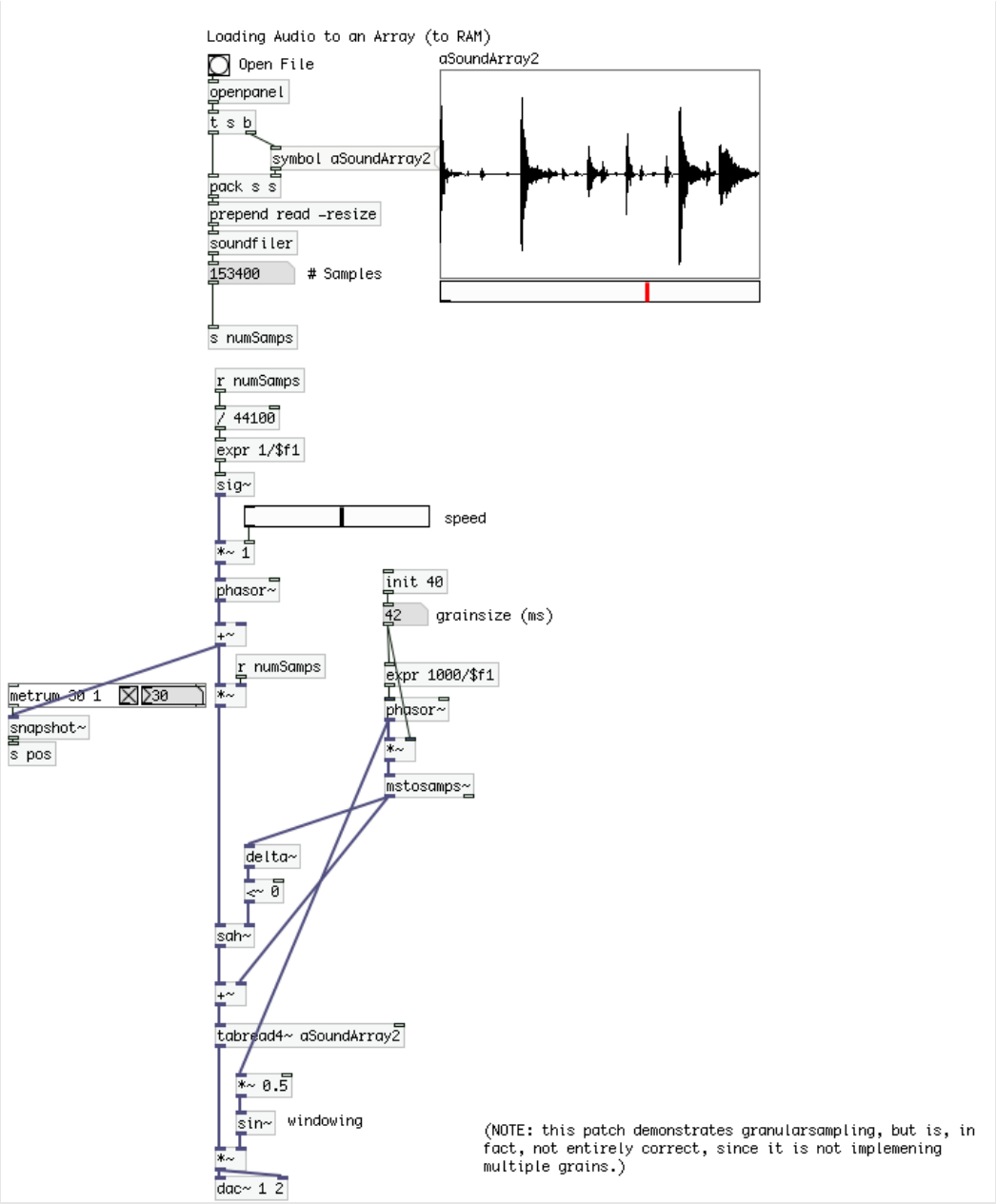


Figure 1.21.: moreSamplimg.pd, a simplified version of granular sampling

1.3. Key Points

- When do we sampling from RAM when from disk?
- what are the problems with waveshaping?
- Make sure you understand that waveshaping can produce overtones.

- Make sure if you see a (simple) mathematical expression you can decide if it is non-linear or not.
- Make sure you recognize waveshaping in pd if you see it (e.g. figure 1.9)

1.4. Hausübung

1.4.1. Testmodul

baue ein audio Testmodul mit folgener spezifikation:

- Ein audio output
- verschiedene klangquellen wählbar:
  1. White Noise
  2. Sinus (freq. einstellbar)
  3. soundfile (file wählbar)
- GUI
- verfügbar(in eurem pfad, und jederzeit abrufbar als abstraction)
- output pegel sichtbar (level meter)

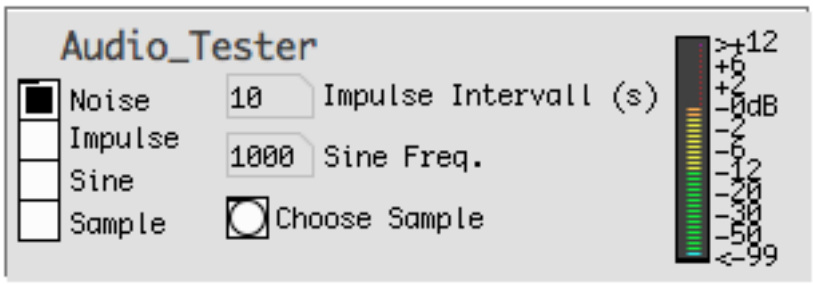


Figure 1.22.: audioTester.pd, zu bauen als Hausübung

2. Modulation

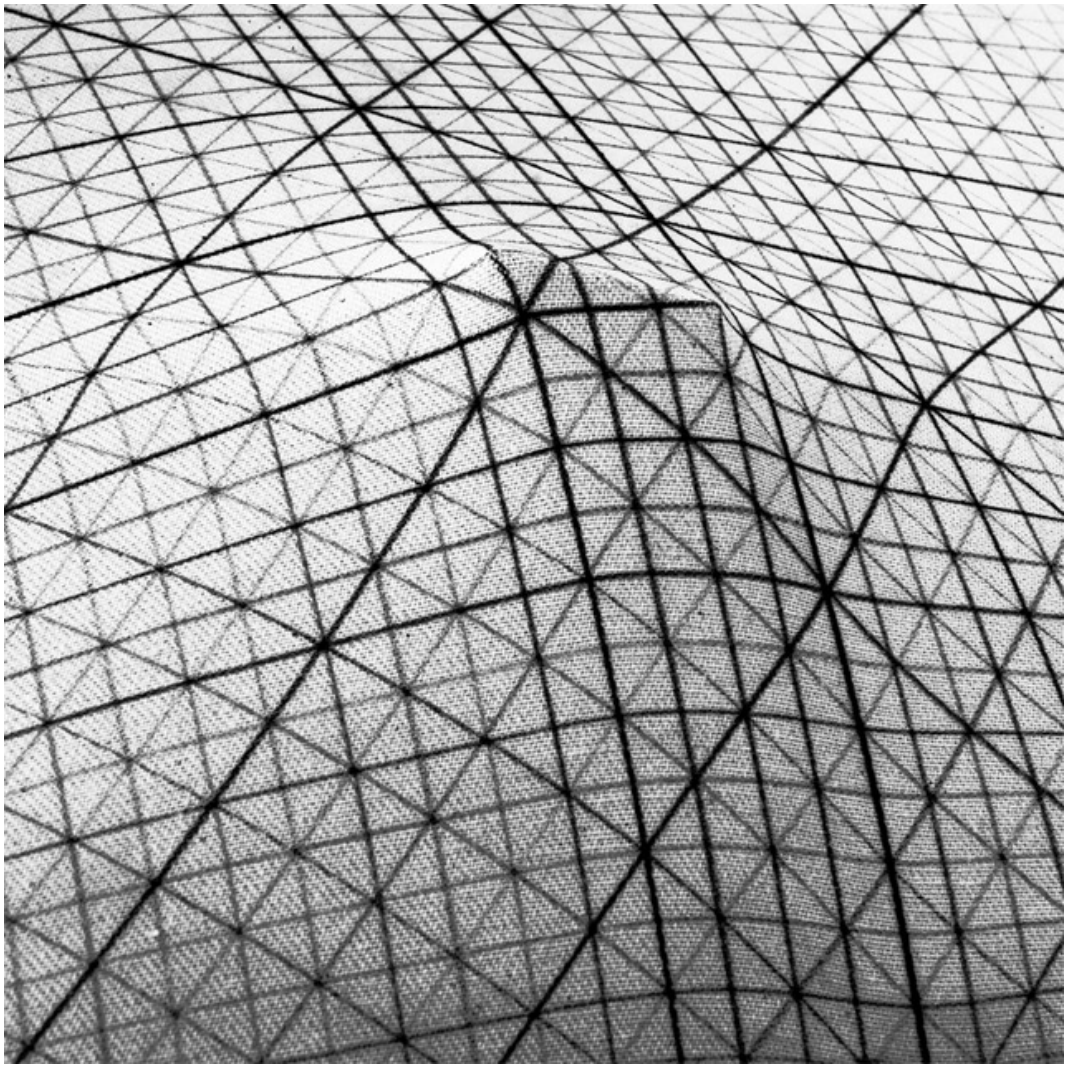


Figure 2.1.: “Surface Modulation” by Richard Sweeney

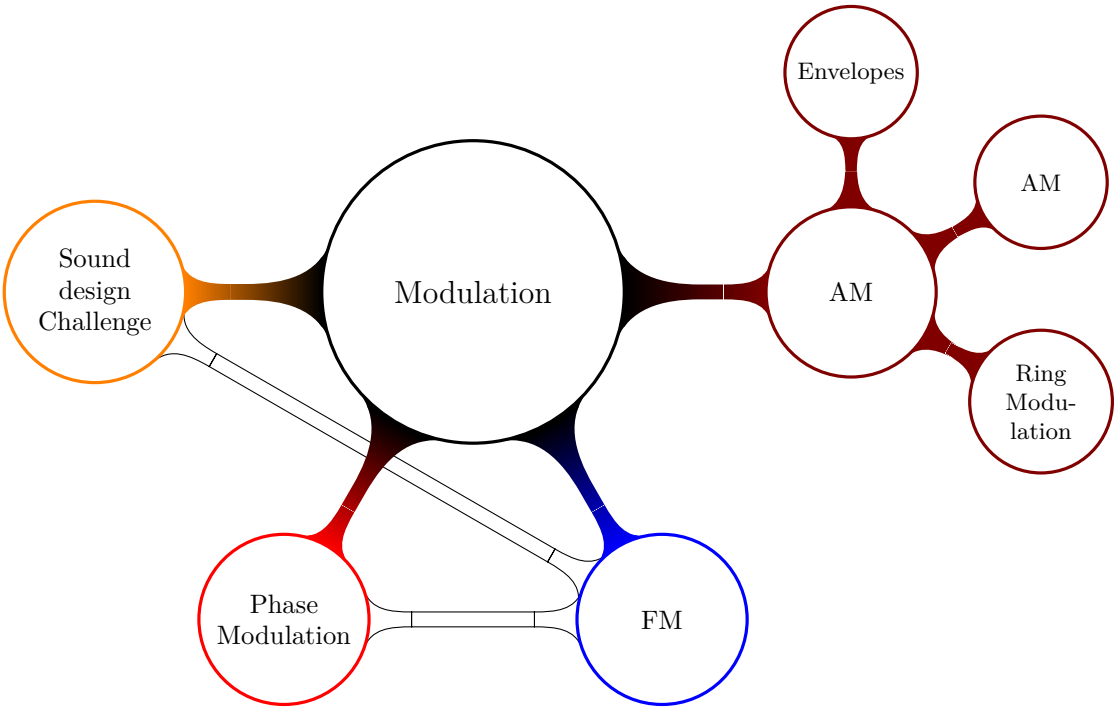


Figure 2.2.: Lecture Contents

2.1. AM

In this section we will try amplitude modulation. Amplitude modulation is used for radio communication (so you'll need to understand this as a technician, modulation techniques are extremely important and this is maybe the simplest one) but also in sound design. We will try to understand the problem from different perspectives at once:

- Doing some math
- listening to it
- brining it in context to beating waves
- seeing it as convolution in the frequency domain

Amplitude modulation means modulating the amplitude of a signal(surprise!). Modulating means changing over time by another signal. So we have some signal, say, a sine wave, and change its amplitude with another signal, say, another sine wave. Talking this concept and reducing it radically, we end up with figure 2.3.

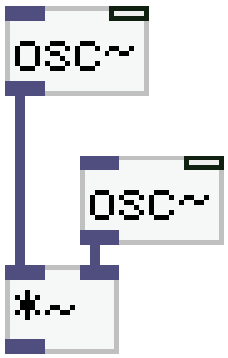


Figure 2.3.: Simplest form of "Ring modulation".

Of course, we get no sound in figure 2.3 because the frequencies are not initialized, but it shows the general principle. The caption of that figure says "Ring Modulation". Let's quickly get some vocabulary straight:

- "Amplitude modulation" might mean any modulation of amplitude
- "Ring Modulation" means *bi-polar* amplitude modulation.
- In sound design, "Amplitude Modulation" might specifically mean unipolar amplitude modulation.

And some more vocabulary to put what we are doing in a musical context:

- Modulating the amplitude is called "Tremolo" in music [1]
- Modulating the pitch or frequency ("FM") is called "vibrato" in a musical context.[2]

Enough words, let's look at what AM looks like, look at figure 2.4.

Question 2 *If we would listen to the signal depicted in the bottom plot of figure 2.4, what do you think we would hear? Try to imagine! If you can't, use pure data to test it! That's why we are using pd.*

Answer 2 *We would hear a 30Hz sine wave repeatedly rising and falling in amplitude.*

[1] sadly, the fender stratocaster's "Tremolo Arm" is used to control the pitch. Ignore Fender, they got it wrong. You can trust that most guitar players are confused because of this.
[2] Maybe think about it like this: The F in FM is a bit like the v in vibrato. Just to avoid confusion..

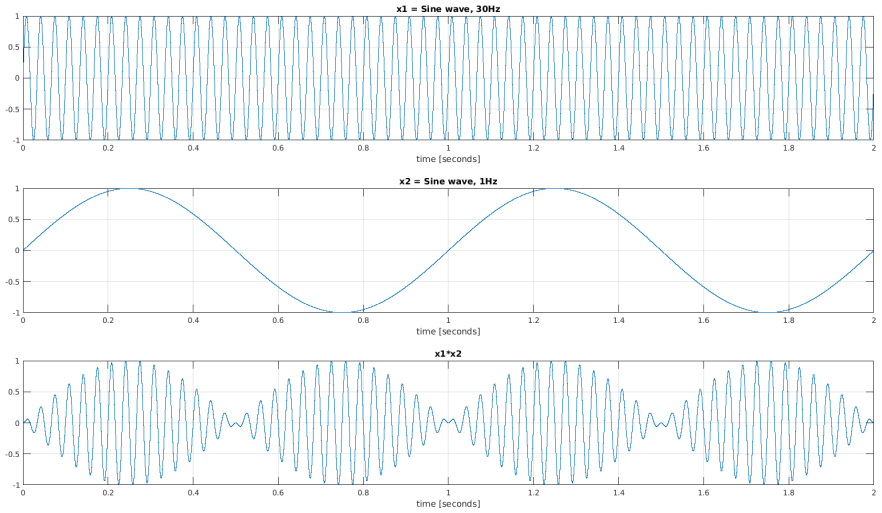


Figure 2.4.: Looking at AM in the time domain

Question 3 *Next question, same plot, same signal. So hopefully you found out that we hear a 30 Hz sine with rising and falling amplitude. At what frequency does the amplitude rise and fall? Remember we are modulating with a 1 Hz sine.*

Answer 3 *Two Hertz.*

Maybe you remember from the waveshaping chapter that we actually calculated what frequencies should come out of AM. Also maybe you remember that:

Amplitude Modulation produces sum and difference frequencies of the input frequencies.

$$cos(a) \cdot cos(b) = \frac{cos(a+b) + cos(a-b)}{2} \tag{2.1}$$

But here we still hear the 30 Hz, just getting louder and softer, so what's wrong? So, let's calculate this. We have two oscillators, $x_1(t) = cos(30t2\pi)$ and $x_2(t) = cos(t2\pi)$. We multiply them, ending up with:

$$y(t) = cos(30t2\pi) \cdot cos(t2\pi) \tag{2.2}$$

Ok, the above formula tells us this means:

$$y(t) = \frac{cos(30t2\pi + t2\pi) + cos(30t2\pi - t2\pi)}{2} \tag{2.3}$$

We can now simplify to:

$$y(t) = \frac{cos((30+1)t2\pi) + cos((30-1)t2\pi)}{2} = \frac{cos(31t2\pi) + cos(29t2\pi)}{2} \tag{2.4}$$

Hm, so we get out a 31Hz and 29Hz oscillator. What about that rising and falling in amplitude that we hear *and* observe in the plot, surely there must be something wrong! Do you have a solution to this?

Let's use pure data to help us understand. Make two oscillators, one with 29Hz and one with 31Hz, what do you hear?

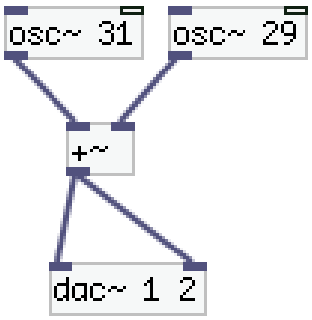


Figure 2.5.: Adding two oscillators with frequencies very close to each other

If you listen to what is depicted in figure 2.5, you will in fact hear the same as if you do the 30Hz /1 Hz amplitude modulation (which indicates that our formula above is correct). What we hear is a phenomenon called beating("Schwebung"). The phase of the two oscillators is canceling each other out at regular intervals because the frequencies are so close to each other. In fact the frequency of the beating f_{beat} is always

$$fbeat = |f_1 - f_2| \tag{2.5}$$

So the difference of the two frequencies.

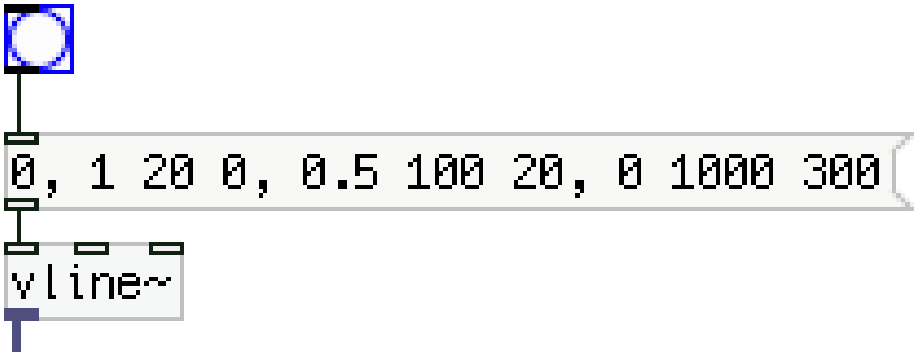


Figure 2.6.: caption

Convolution We will look at convolution in a later chapter. If you are eager to know what convolution exactly does you can have a look at this[a] . But why mention convolution if it is not explained at this point? Amplitude modulation and convolution just have a very close relationship, an that is: **Multiplying two signals in the time domain is equivalent to convolution in the frequency domain and convolution in the time domain is equivalent to multiplication in the frequency domain.**

[a] https://youtu.be/_vyke3vF4Nk?t=25m14s

2.2. FM

Frequency modulation can be used to modulate the frequency, meaning to vary the pitch of a sound, see figure 2.11. But it can also be used to generate overtones and an overall richer spectrum, see figure 2.12. The two "different versions" only differ from each other by different parameter[3] values being used. We can think of Frequency Modulation (FM) as something of the form:

$$y(t) = cos(cos(b)) \tag{2.6}$$

This really is a simplification, but the overall structure of the formula is correct. Looking at figure 2.7 we can see the very basic idea implemented in pure data.

[3] We see which parameters can be adjusted in a minute. But if you want to know them now: modulation frequency, modulation amount and carrier frequency.

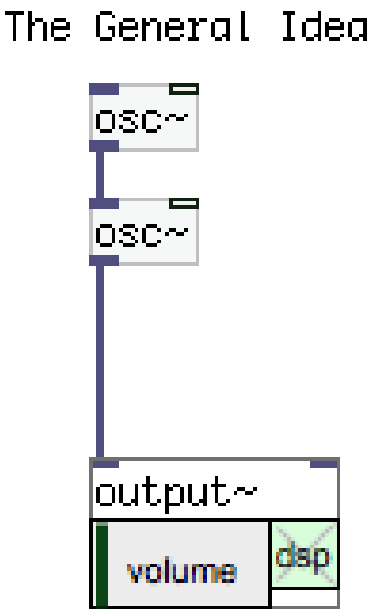


Figure 2.7.: The General Idea of FM

Since the frequencies of the oscillators are not set, we won't hear anything when building this patcher. But looking at it may reveal that the idea simply is to control the frequency of an oscillator using another oscillator.
We can expand the patcher by adding some math to make it more usable, as in figure 2.8.
Naive parameters are f_c (Carrier Frequency), f_m (Modulator Frequency), and A_m (modulation Amount).

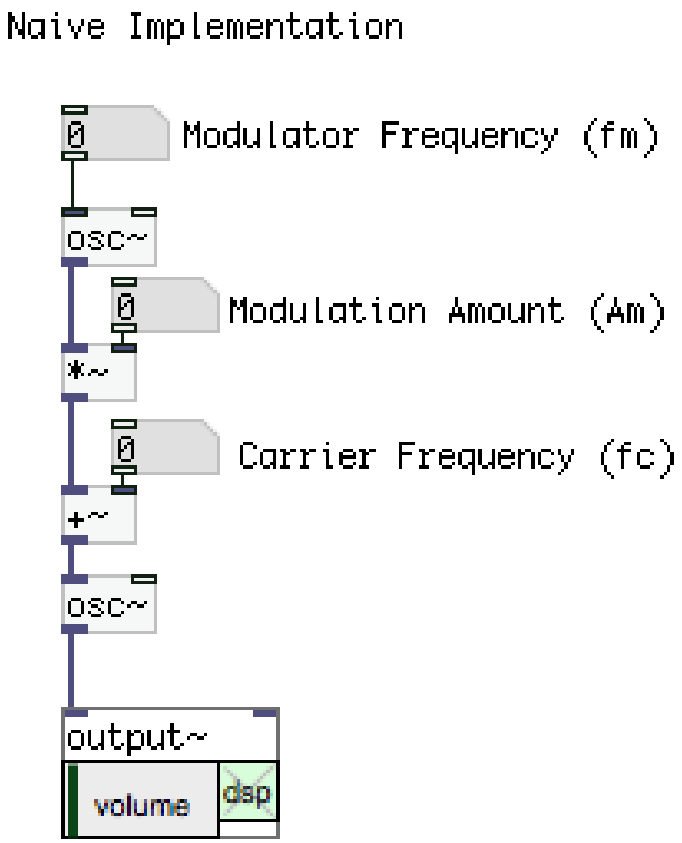


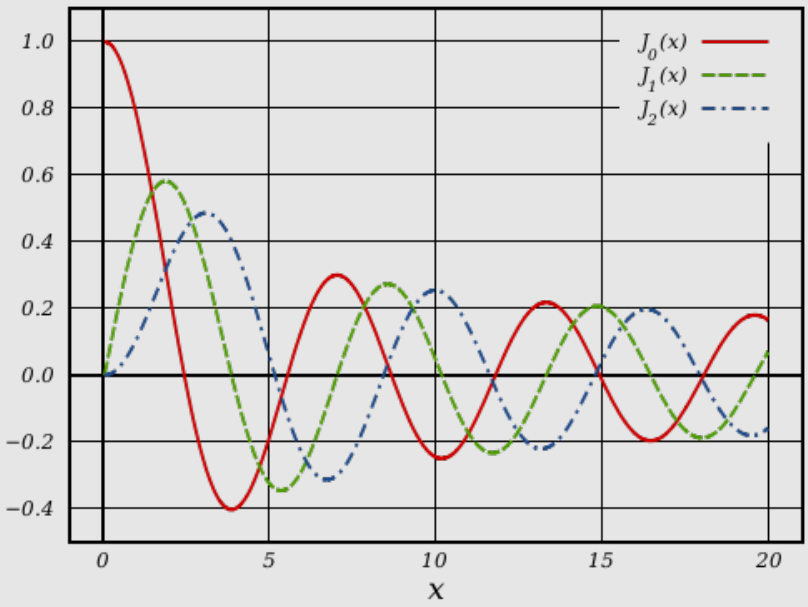
Figure 2.8.: Naive Implementation with Direct Parametrization.

The output frequencies will be

$$f_c \pm n \cdot f_m \tag{2.7}$$

for n being all integers. So the carrier frequency plus and minus integer multiples of the modulation frequency. We get many (theoretically infinitely) many overtones with different amplitudes this way.

The amplitudes of the different frequencies are determined by Bessel functions which makes it so seemingly random.



Typically, FM is controlled via *Index*, *Ratio*, and fundamental Frequency. The Index, I is given by Modulation Depth and Modulator Frequency.

$$I = \frac{A_m}{f_m} \tag{2.8}$$

A more controllable Implementation will generate the naive parameters from a Ratio, R, the Carrier Frequency and the Index:

$$f_m = \frac{f_c}{R} \tag{2.9}$$

$$A_m = I \cdot f_m \tag{2.10}$$

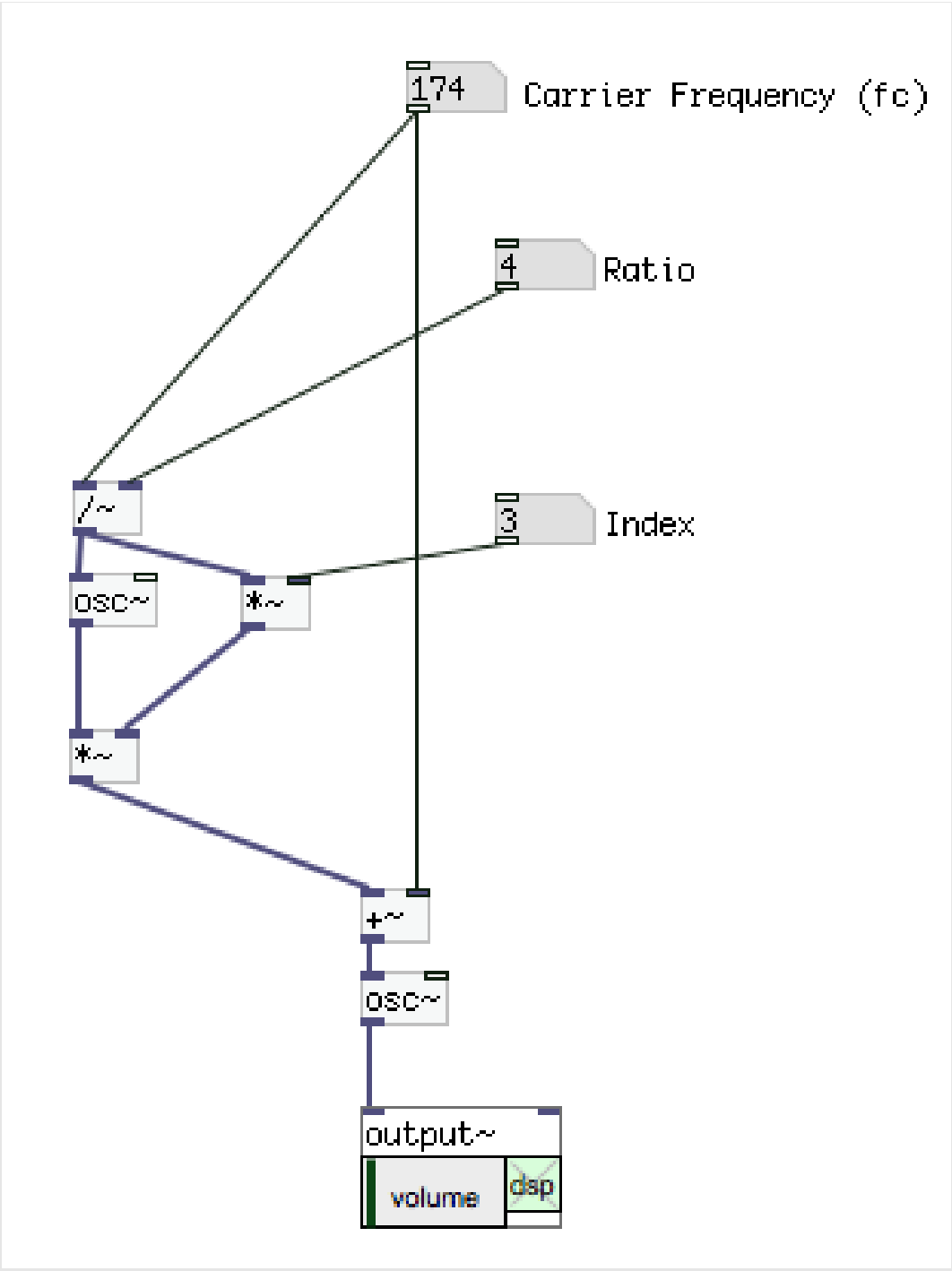


Figure 2.9.: FM with Index and Ratio

But there are also different implementations such as:

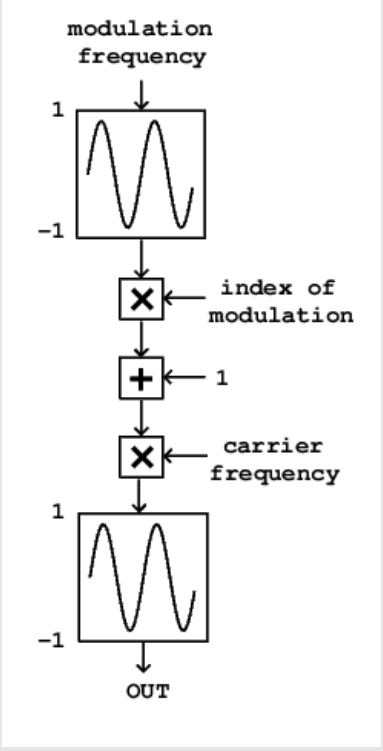


Figure 2.10.: Fm implementation from Miller Puckette (2006)

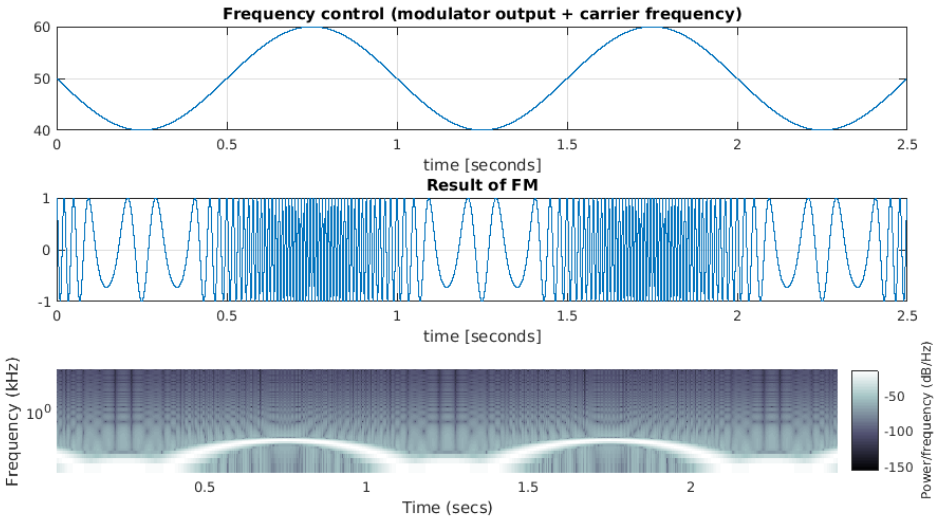


Figure 2.11.: Modulator frequency: 1Hz, Crarrier Frequency 50 Hz, Modulation Amount 10. Please ignore the labeling of the Y axis of the spectrum plot(10^0). It is wrong. The y axis goes from 0 to nyquist=22050Hz.

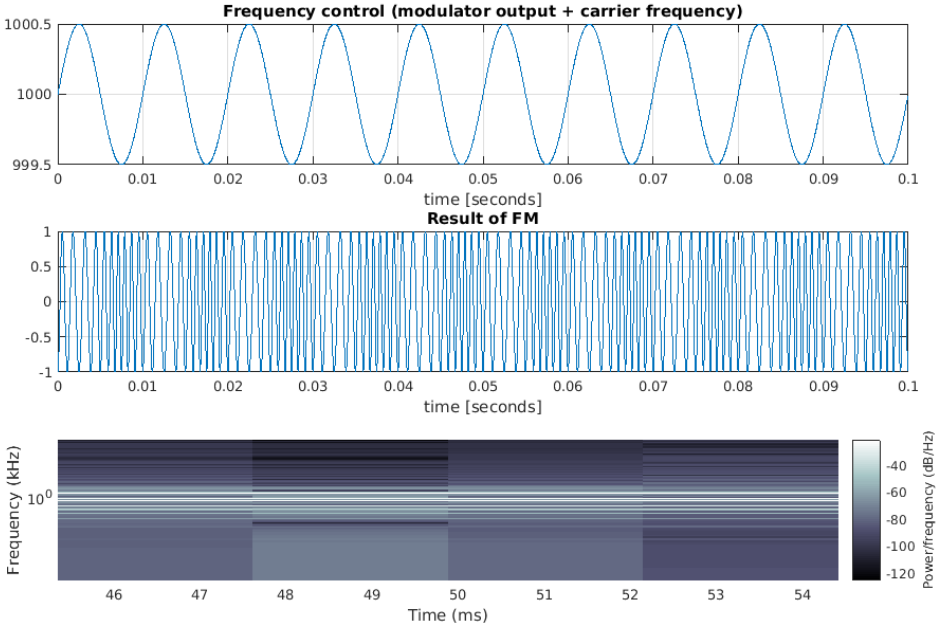


Figure 2.12.: Modulator frequency: 100Hz, Crarrier Frequency 1000 Hz, Modulation Amount 0.5. Please ignore the labeling of the Y axis of the spectrum plot(10^0). It is wrong. The y axis goes from 0 to nyquist=22050Hz.

2.3. Key Points

- Make sure you understand the differences between AM and FM
- Make sure you recognize what form of modulation is present if you see a patcher or a simplfied formula (like: $sin(a) \cdot sin(b)$ what form of modulation is this?)
- make sure you could make such a simplified formula if you see a patch and vice versa.
- make sure you know what frequencies result from an amplitude modulation
- make sure you know what frequencies result from a frequency modulation

3. Filters

Figure 3.1.: Caravaggio, Narziss. ad Feedback: Echo liebt Narziss etc.

lecture mind map missing

What is a filter? We use filters a lot. We often need to shape the spectrum of a signal. Be it for technical reasons (DC-Offset removal, Anti-aliasing filters, crossovers, interpolation, ...) or for aesthetic ones (EQ-ing, subtractive synthesis)
Please take a moment and seriously ask yourself, knowing what you know about signals, knowing what we can do with a computer: How do we actually do this? What is a digital Filter? This is what this chapter will try to explain.

3.1. Seeing Frequency Content in the Time Domain

Let's first try to get a feel for how signals *look*. It's a lot easier to understand how a filter works afterwards. This might be obvious but let's state clearly:

> High frequency signals have a short *period*. That means the signal "moves" fast, or a lot in short time. Therefore: The more movement is in a signal or the more fluctuation or the faster the signal changes the more high frequency content it has.

Three Questions in order to make you think about signals:

Question 4 *Please look at the image below. Can you draw an approximation of its spectrum? What signals could have been mixed to obtain this result? To help you think about this: How would you explain the graph to somebody who doesn't see it?*

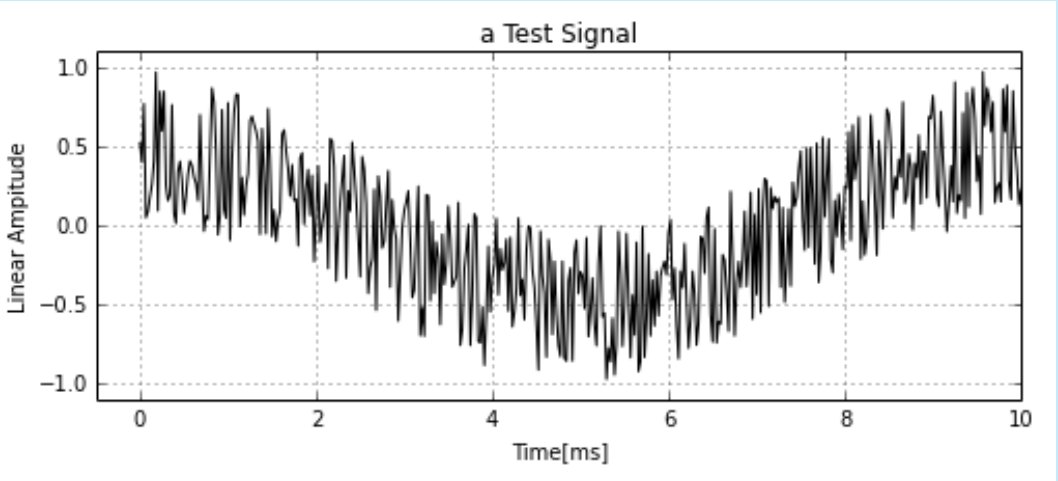


Figure 3.2.: Original Signal

Answer 4 *You should be able to see that there is a sinusoidal component in the signal. This sinusoid has a period of 10 ms, so a frequency of 100 Hz. But also you should see that it is not a clean sine wave, there is some noisy component in there. This is what you should have been able to infer from the image. In fact it is an (attenuated) 100Hz sinusoid mixed with (attenuated) white noise.*

Question 5 *The signal in figure 3.2 was filtered. As a result we obtained the signal in figure 3.3. What kind of filter could have been used?*

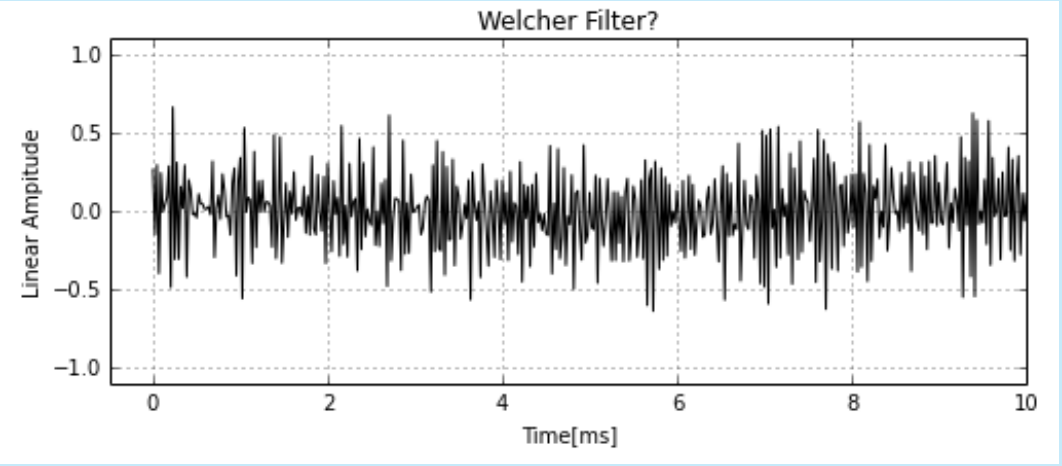


Figure 3.3.: Filtered Signal 1

Answer 5 *It could have been a notch or stopband filter at 100 Hz or a highpass. In fact, it was a highpass.*

Question 6 *Again, the signal in figure 3.2 (so the original signal again) was filtered. As a result we obtained the signal in figure 3.4 . What kind of filter could have been used?*

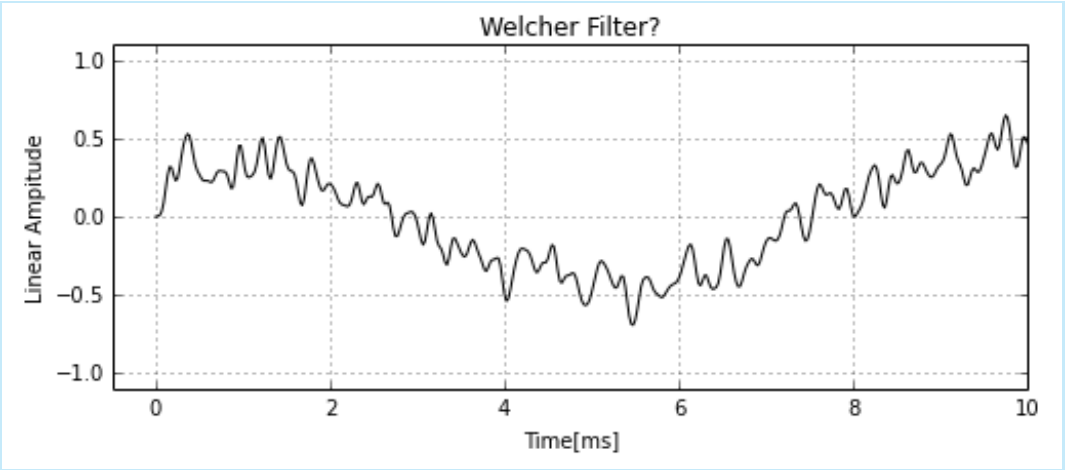


Figure 3.4.: Filtered Signal 2

Answer 6 *It could have been a bandpass at 100Hz, or a lowpass. In fact, it was a lowpass. We see that the signal got a lot smoother.*

3.2. Ways of Describing a Filter

There are a couple of ways to *fully* describe a linear time invariant system (LTI), filters typically[1] are such systems.
Below is an (incomplete) list of methods that are all able to describe a filter. They offer very different possibilities of understanding and manipulating a filter.

- Difference Equation
- Magnitude and Phase Response
- Impulse Response
- Code (graphical, such as pd)
- Code (text, such as C++)
- Block Diagram
- Transfer Function (Rational Function)
- Unit Step Response

[1] filters can be augmented with non-linear or time-varying elements, for example distortion or modulation. In sound design, this is actually very common. But describing such a system mathematically is out of the scope of this work.

While in practice, the transfer function is very important, we will mostly work with block diagrams, pd programs and difference equations.
It is an important skill to be able to switch between these representations, for example to calculate the impulse response from a difference equation or block diagram.

3.2.1. Difference Equations

A difference equation is of the form mentioned in the introduction. So for example

$$y(n) = x(n) + x(n-1) \tag{3.1}$$

Here, y is the output x is the input and n denotes indexing those two signals.
$y(n) = x(n)$ means: take the incoming sample and make it the output. $x(n-1)$ denotes a delay by one sample. Please note that the above equation describes a *system*, a filter in this case. But it can be seen as using an input array (the input signal), indexed by n and creating an output array.
Let's visualize this idea. Given the input signal in figure 3.5, we can create the output using the equation. Also it becomes obvious that the $n-1$ expression is a delay, since it makes us look up the 3rd index for creating the 4th sample at the output in this case, or $y(4) = x(4) + x(4-1)$.

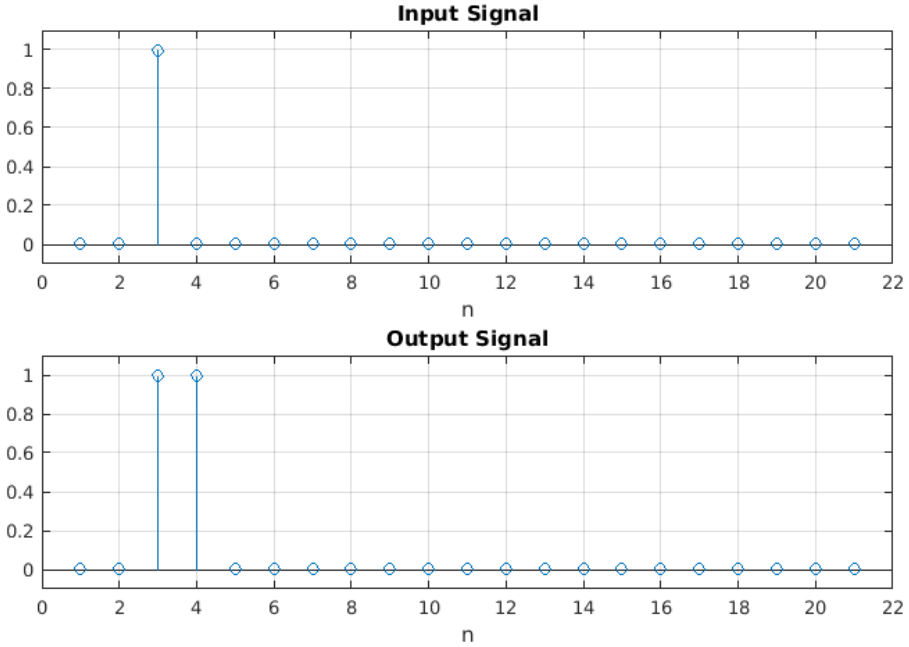


Figure 3.5.: The input and corresponding output of the equation given above.

3.2.2. Block Diagrams

Block diagrams are nice because they are very visual and sometimes easier to understand. Also, if we work in pure data, or a different graphical programming language (such as Max/MSP or MATLAB Simulink) it's very straight forward to implement a block diagram. Translating from a block diagram to text code or difference equations or vice versa can be hard sometimes.
In figure 3.5, we can see a block diagram representation of equation 3.1. This *is* the same thing.

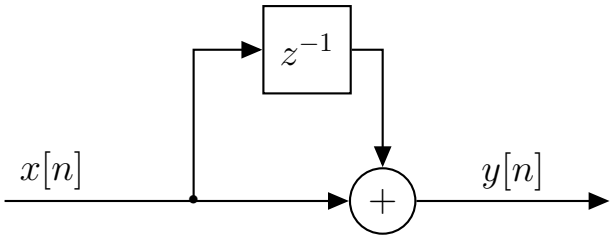


Figure 3.6.: A very simple lowpass filter.

Don't be irritated by the z^{-1} block. This notation stands for a delay by one sample. A block z^{-m} would stand for a delay by m samples. This is a convention we need to remember. This way of notating a delay might seem unnecessarily complex, but it has its reasons.

This notation, z^{-1}, comes from the fact that analyzing the frequency response of a system is very easy when working with complex numbers (so compounds of real and imaginary numbers). Complex numbers are often denoted by Z, which is another convention. Not only do we work with complex numbers, but with complex sinusoids when analyzing such systems. We imagine complex sinusoids coming into our system. A multiplication by another complex sinusoid can cause a phase shift (a delay). Dividing ($z^{-1} = \frac{1}{z}$) an incoming complex sinusoid by the complex sinusoid at the same frequency causes a phase shift that is equals to a delay by 1 sample. If you want to understand all this and stop with mickey-mouse-explanations I can recommend reading Miller Puckette (2006) and Smith (2007). Both of these works can be found on-line for free.
But let's quickly calculate the frequency response, so you've seen it once: We take our difference equation (we could also start from the block diagram):

$$y(n) = x(n) + x(n-1) \tag{3.2}$$

We then do a process called the z-transform, that is, we get rid of all the indexing. We now think about applying the process of filtering to the whole signal at once

and we denote the delay by a multipliaction with a complex sinusoid that causes a phase shift:

$$Y = X + X \cdot Z^{-1} \tag{3.3}$$

We then solve the equation for Y/X:

$$Y = X \cdot (1 + Z^{-1}) \tag{3.4}$$

$$\frac{Y}{X} = 1 + Z^{-1} \tag{3.5}$$

Typically we substitute $\frac{Y}{X}$ with a new name, typically H and now make this a function of Z.

$$H(Z) = 1 + Z^{-1} \tag{3.6}$$

This is called a system's transfer function. This is the actual standard for talking about systems. Typically we do two things with this, either, we draw a so called *Pole-Zero-Plot*, or we evaluate the function for complex sinusoids.
The absolute value of H is the output amplitude, if we just plot this value for all complex numbers we see this:

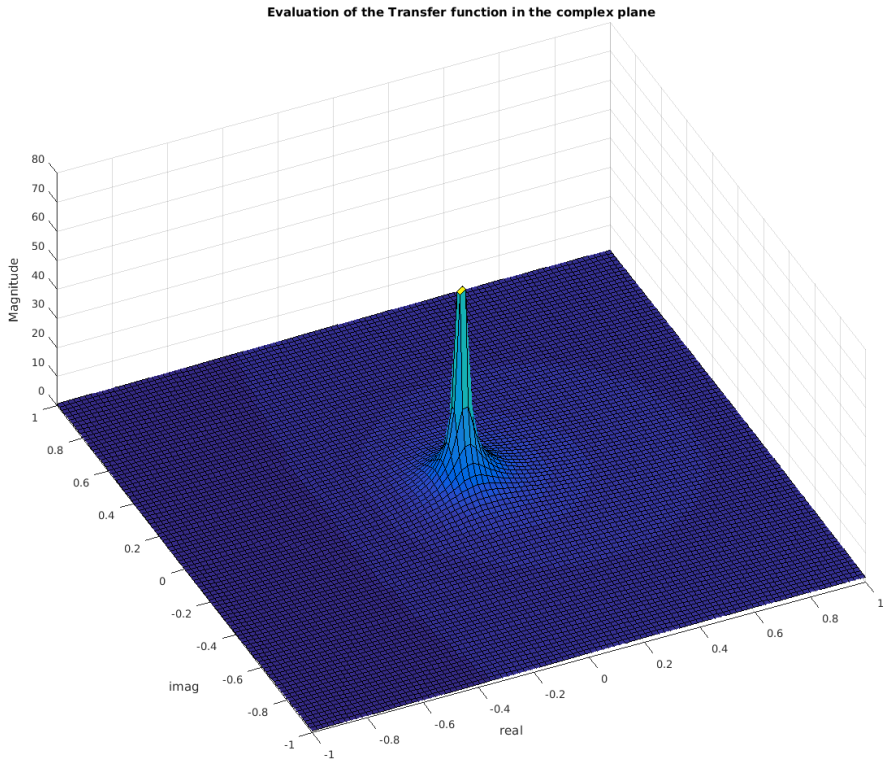


Figure 3.7.: CAPTION MISSING

It's OK if this plot doesn't tell you much. The really important thing is to evaluate this surface using complex sinusoids: $cos(\omega) + i \cdot sin(\omega)$ we can now plot the magnitude response over ω. So we effectively evaluate the formula

$$|H(\omega)| = |1 + cos(-\omega) + i \cdot sin(-\omega)| \tag{3.7}$$

Where ω ranges from 0 to π(sic!). Then we finally get the magnitude response:

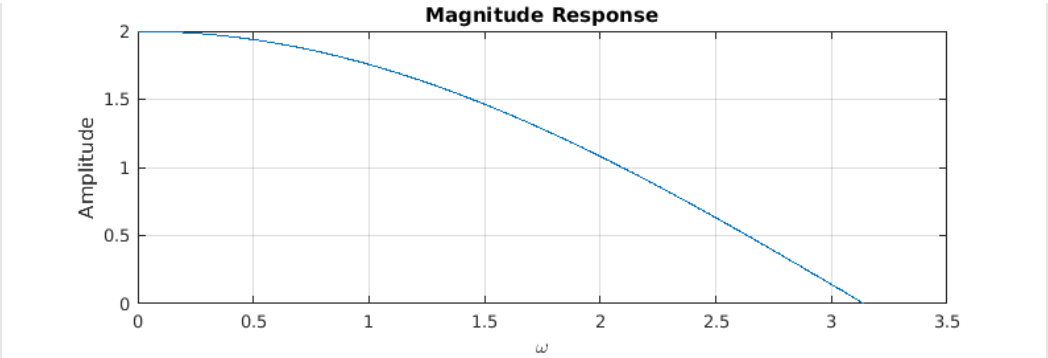


Figure 3.8.: CAPTION MISSING

3.2.3. pd Code

If we program the filter in pd (or any other programming language) we obviously also fully defined the filter. Try to build the filter from above in pd! Start from the difference equation or the block diagram, what ever is harder to understand for you.
In figure 3.9 you can see the translation to pd.

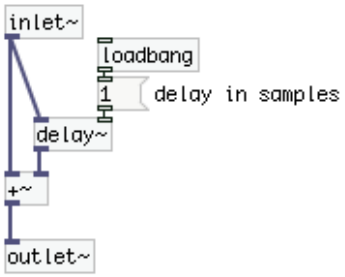


Figure 3.9.: A simple filter in pure data.

3.2.4. Impulse Response

An impulse has the advantage that it contains all frequencies. It has a flat spectrum. So recording how a filter reacts to the impulse gives us the magnitude of the spectrum at all frequencies.
We can see the impulse response of the filter we are describing in this section in multiple ways in figure 3.5.

3.2.5. Text Oriented Code

Below, we can see a code implementation in the language MATLAB.

```
%% setup
```

```
N = 20 %number of samples to compute
imp = zeros(N,1); %creates a list of zeros, length N
imp(3) = 1; % sets the third sample to one. So we have an impulse.

%% processing
x0 = [imp;0]; %takes the input and append a zero.
x1 = [0;imp]; %takes the input and prepends a zero(= delay input 1
    sample).
Y = x0+x1; %computes the output signal.

%% Plotting
% plot input
subplot(2,1,1)
stem(x0)
xlabel('n')
ylim([-0.1, 1.1])
grid on
xlim([0,N+2])
title('Input Signal')
% plot output
subplot(2,1,2)
stem(Y)
xlabel('n')
ylim([-0.1, 1.1])
xlim([0,N+2])
grid on
title('Output Signal')
```

Listing 3.1: A script that creates an impulse implements a vectorized filter and plots the result.

3.3. Ways of Getting an Intuitive Understanding

So now we learned some ways to define a filter, to talk about a filter. But what is this filter actually doing? It is a very simple lowpass filter. It cuts away high frequencies. But why is adding a signal to itself but one sample delayed making a lowpass? This certainly is a bit surprising, so let's try to understand what's happening.

3.3.1. Combfilter to Lowpass

First let's view the problem from another angle. We know what this structure from above is: It's a combfilter, right? It's taking a signal and adding a delayed version of it. Let's quickly review what a combfilter is: The combfilter effect can come up if we record something with a microphone and we get the direct signal and a delayed (e.g. via a reflection) signal also. These two signals mix together and this is what we get in our recording.

The result is that some frequencies cancel out.
We can simply simulate this situation in pd using a delay and an addition.

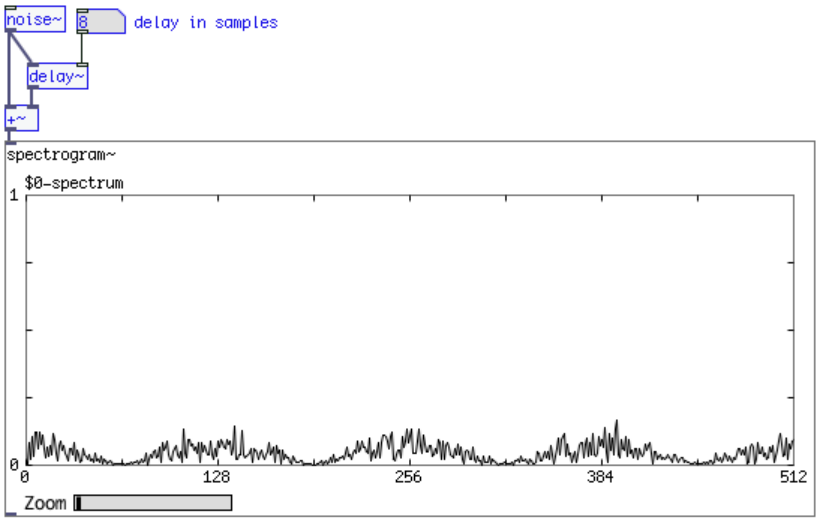


Figure 3.10.: realCaption

This canceling out of frequencies [2] can be imagined if we look at figure 3.11.

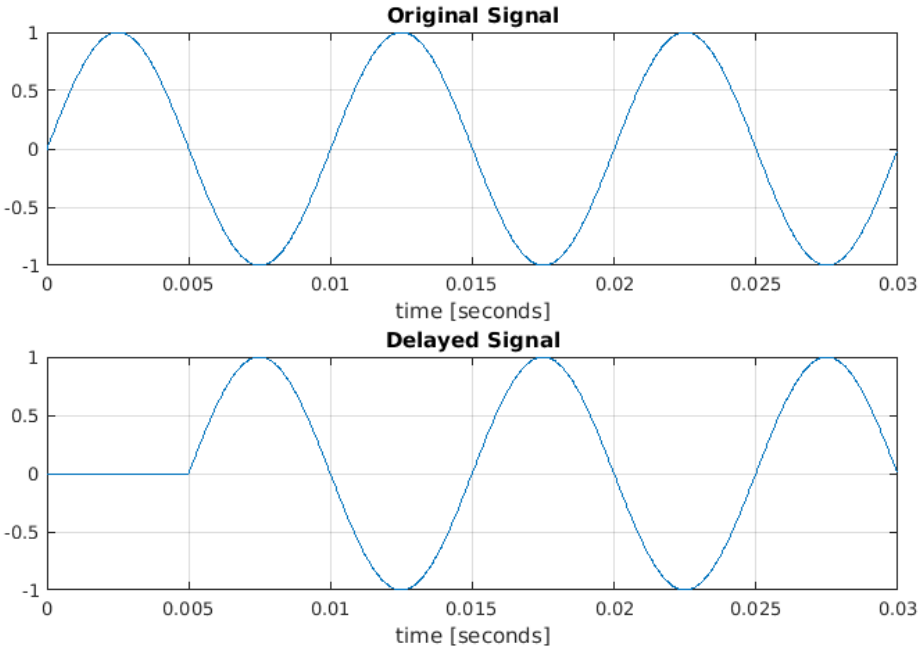


Figure 3.11.: Destructive Interference. The two waves would cancel out completely if mixed together(=added).

[2] also called destructive interference

Please note that the signal is a sine wave with a frequency of 100 Hz, so a period of 10 milliseconds. The delay is set to be 5 ms, so half of the period of the input signal in this case. It it was set to be equals the period, the two waves would *interfere constructively*, so we would get out a higher amplitude.

In a combfilter, the first frequency that cancels out is the one whose period is double the delay time. Or, to put it differently

$$f_c = \frac{1}{2d} \tag{3.8}$$

If f_c is the first frequency that cancels out and d is the delay time in seconds.
How to calculate this fast, if you are given the delay time: just calculate the frequency for the given delay, and then use half of that. For example, if given delay 1ms, the corresponding frequency would be 1000 Hz. Half of that (=double the period) is 500 Hz, and that's where the first trough in the spectrum would be.

In figure 3.12 we can see the *frequency response*[3] of our combfilter. We see a couple of plots, it's always the same filter but the delay is different. We see that for delay times >1 sample, we observe our typical comb pattern. But at Delay = 1 sample, there is just a ramp left, leaving us with a lowpass.

[3]The impulse response is recorded a couple of times and the output's spectrum has been analyzed

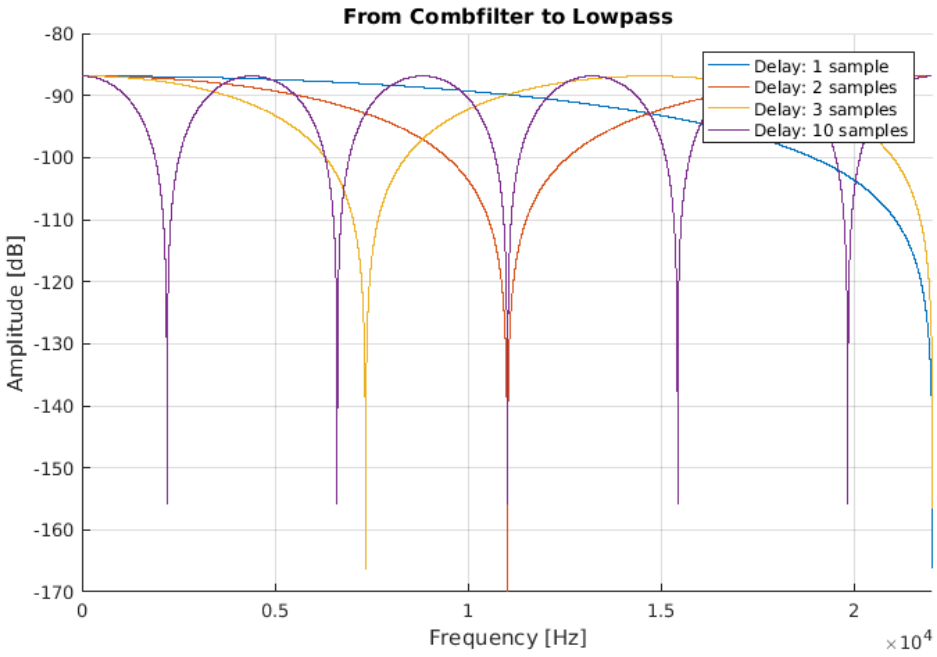


Figure 3.12.: realCaption

We can also calculate what's happening with our equation from above: The delay is $1/f_s$, so 1 over samplerate. This is called the sampling intervall, let's call it I_s, just to get rid of the fraction. The frequency that cancels out is then $1/2I$. By inspection we find that this means the frequency that has double the wavelength of the sampling frequency, so half the sampling frequency, so *Nyquist*.

3.3.2. Prolonging an Impulse

We can also take a different route to understand why this is a lowpass. Let's recall what the spectrum of an impulse looks like and what the spectrum of DC offset looks like. Taking a look at figure 3.13, we see an impulse, the impulse response of our filter and a DC-Offset signal. Recalling from the introduction, and looking at the figure, we know that the DC-offset signal contains only energy at 0Hz. Now look at the time domain signals: Isn't making our impulse signal last for one sample longer a small step towards generating a DC-offset signal?

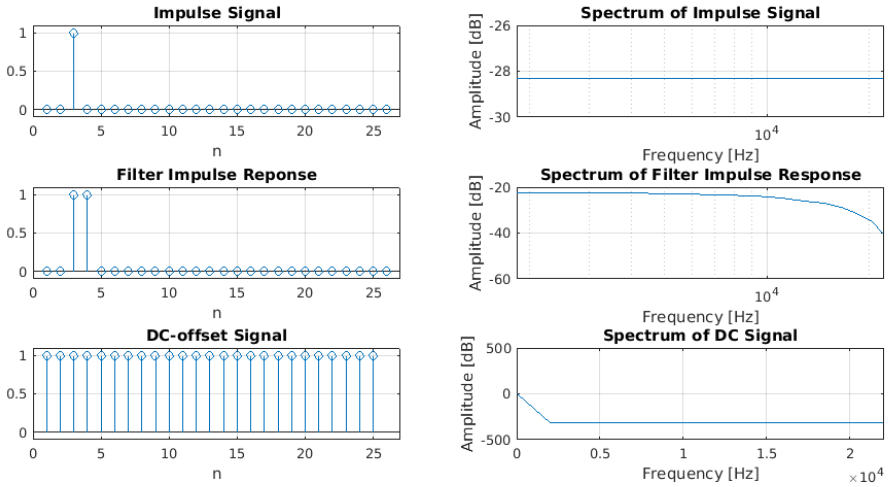


Figure 3.13.: Impulse, impulse response and DC offset signal and their spectra. Note that there are very few samples here and the DC-Offset signal's spectrum seems to ramp down from 0Hz. In fact, there is a single high value at 0Hz.

3.3.3. A moving Average

How is *average* defined? There are a lot of different ways to compute an average, but typically we mean the *arithmetic mean*, so adding all numbers and dividing by the count of items:

$$A = \frac{1}{N} \sum_{n=1}^{N} x(n) \tag{3.9}$$

Looking back at our filter, we will notice that we are adding up the current sample and the last sample. If we were to divide that sum by two (so the number of elements, since it's two samples we are adding up), we would arrive at the definition above. So we could say, that our lowpass is always outputting the average of the current and the last sample. It is therefore called a moving average filter.
Taking the average of something always reduces information and we get something that is fluctuating less than its input. If we would make a website or something that would always show the average temperature in Vienna of the last two weeks, this display wouldn't start jumping around if there was a single hot day in winter. It would remove high frequency data (jumps). It would be some kind of lowpass filter.

3.3.4. A sine at Nyquist

Here is yet another way of understanding why the structure we saw is a lowpass. Let's imagine a sine wave at Nyquist frequency, the highest frequency we can work with. We

can see what such a sine looks like in Figure 3.14. If we apply this moving average filter, so that lowpass we have been talking about, what do you think comes out? Go ahead and calculate the average for each sample and its previous sample. We add $1+-1=0$, $-1+1=0$, $1+-1=0$, ... So you can see, if we feed a sine at nyquist into our lowpass, what we get is silence.

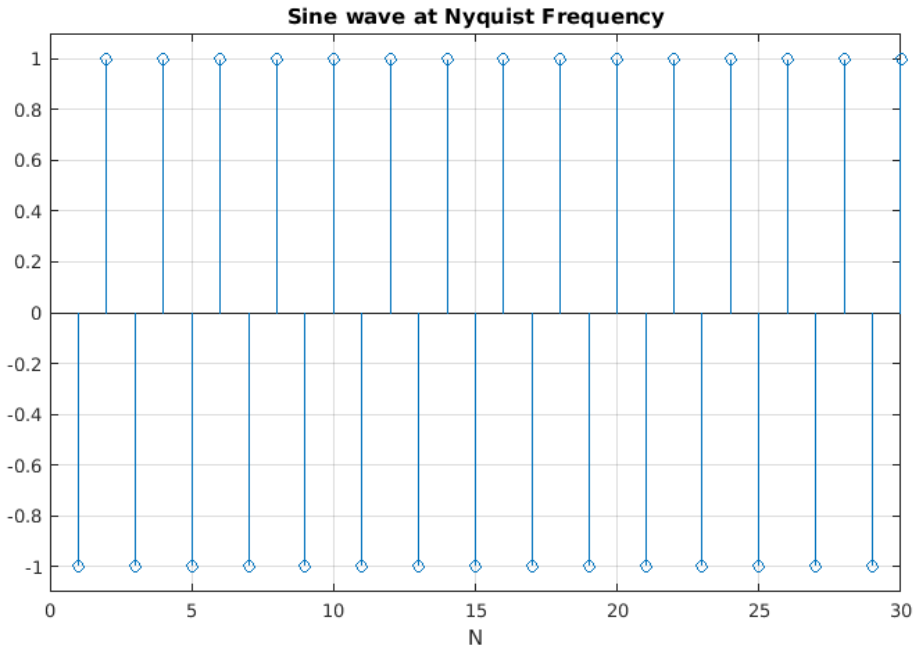


Figure 3.14.: CAPTION MISSING

3.3.5. Highpass

You should already be familiar with the basic types of filters, so lowpass, highpass, bandpass etc. We now saw how we can build a lowpass filter, but what about a highpass, what about notching, peak, shelving filters? This is getting pretty involved quickly, but let's have a look at a highpass. Highpass filtering is the opposite of lowpass filtering. It emphasizes movement in the signal, since everything that moves fast must contain high frequencies. One simple form of a highpass is taking the difference between the current and the last sample, so:

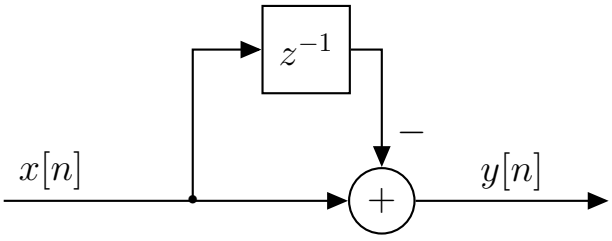


Figure 3.15.: A very simple highpass filter. Please compare to 3.6.

You will notice that figure 3.15 and figure 3.6 are identical with oly one little exception. In this highpass filter we **subtract** the delayed signal from the input. This is denoted by the little − sign next to one line leading to the addition. One might think, "Why don't we just use a minus sign instead of the addition sign?" Because order matters with subtraction, it makes a difference if we subtract 3 from 5 or 5 from 3.
The structure above is also called a differentiator, since it is a way of approximating the derivative of a signal.

3.3.6. Video Filters

Video Analogies
Filters in video work the same way and are as common as in audio. But in video we have another question to answer before we start filtering: Does our operation (adding two neighboring samples) apply to time, or to space? In other words: Do we want to add neighboring pixels to produce an output pixel or do we want to add whole frames over time to produce an output frame?
Have a look at figure 3.16. We will take this still from a video as a reference. We send the video through a couple of filters (one at a time, not in series) and watch the output.
For each of the output frames, try to think about how this could have been generated from the input, what kind of a filter could have been used!?

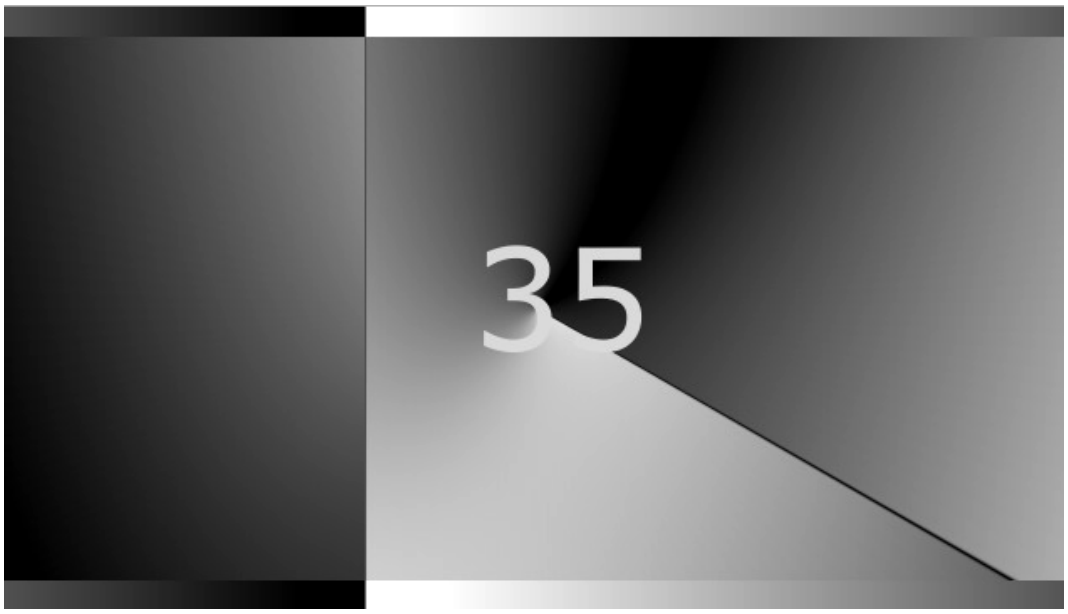


Figure 3.16.: Frame from the original video

Figure 3.17.: This effect is known as a *blur* It is generalted by using a FIR lowpass in the space domain. That means, in the simplest case, each output pixel is the average of the input pixel at the same location and its neighbors. In practice, weighting functions are used to control, how much the input pixel and how much its neighbors contribute. These weighting functions are called kernels, (since this FIR filtering is the same as convolution). Typical kernels are: Gaussian, Box, the sinc function, etc.

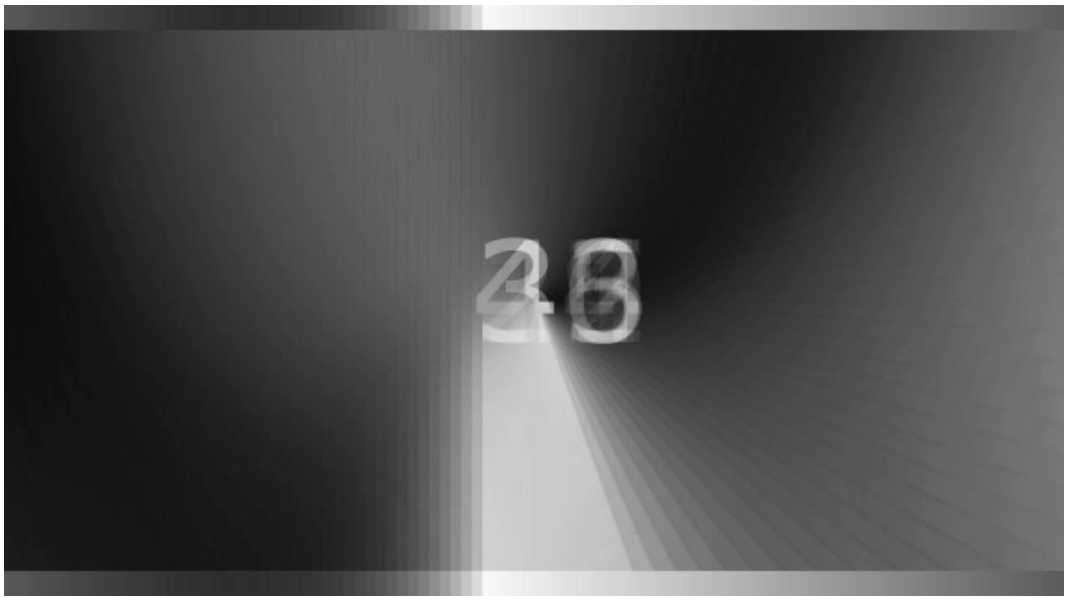


Figure 3.18.: Here we can see a kind of lowpass (IIR) in the time domain. The current input-frame and a mixture of past (output-)frames are added up to generate this output frame.

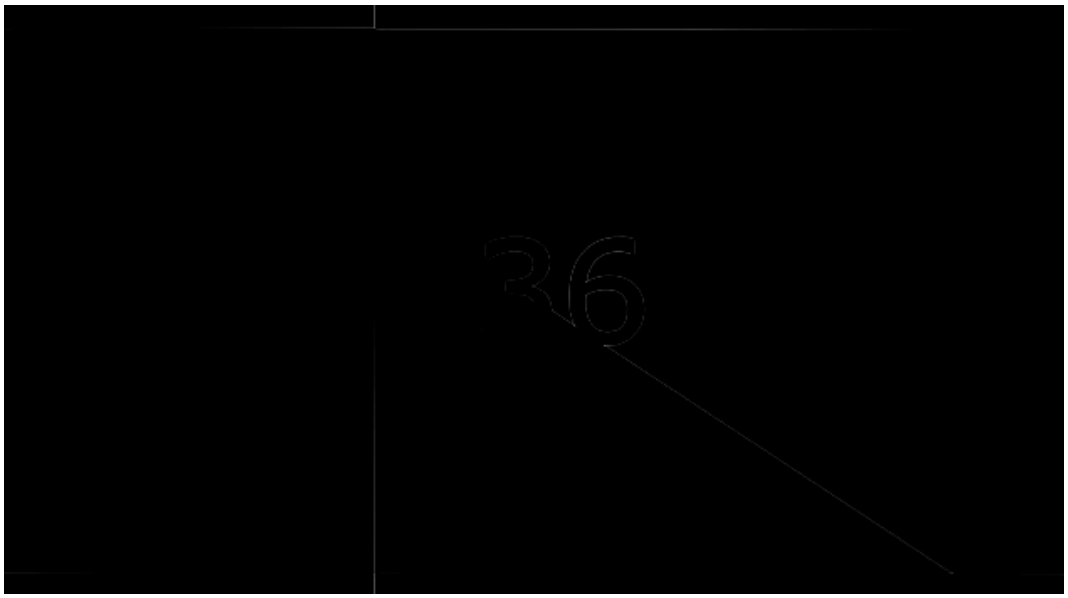


Figure 3.19.: We can hardly see anything here, but if you look closely, edges from the original are still visible. What happened here is a FIR filtering in the space domain, so the opposite of blurring: Edge detection.

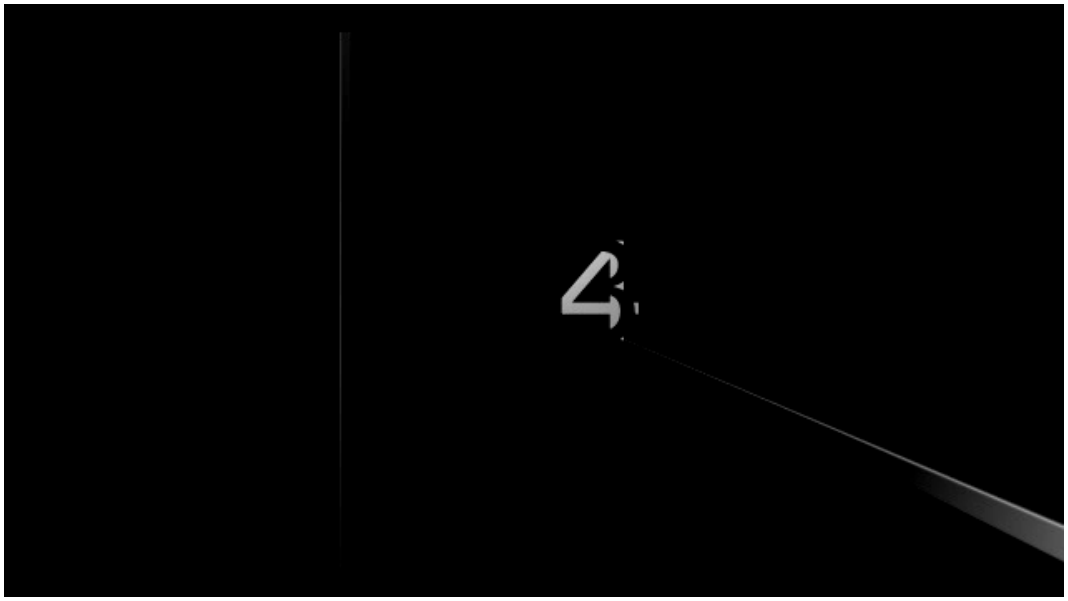


Figure 3.20.: What we can see here is another highpass, but in the time domain. It computes the difference between the current and the last frame.

3.4. Filters in pd

Please note that what we did so far was filtering signals, but we **built** the filters with very low level operations (Addition, multiplication and time shift = delay). This was in order to understand how filters work. Sometimes we need to build a filter, but often we can just *use* filters, which are built in to pd or any other environment.

3.4.1. Lowpass

In pd, we can use `[lop~]`, `[onepole~]`, `[lores~]` and `[vcf~]`. `[lores~]` has a resonance parameter. This emphasizes frequencies at the cutoff frequency of the filter. `[vcf~]` also has this parameter and an additional bandpass output.

3.4.2. Highpass

There is `[hip~]`, which is a highpass filter. But we can just use a structure that subtracts a lowpassed version from our original signal also. For example:

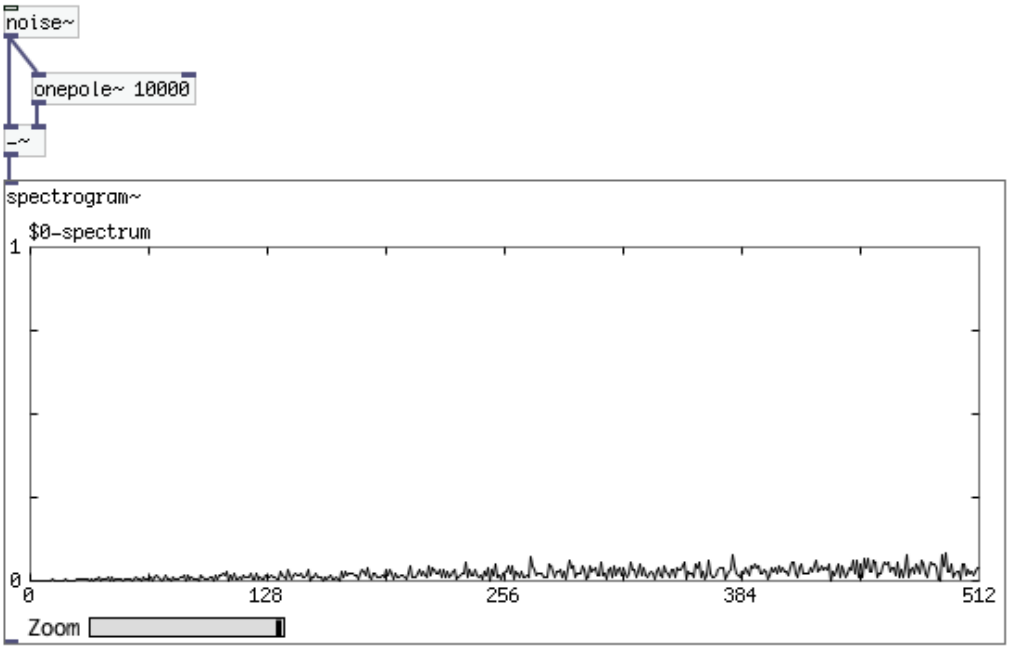


Figure 3.21.: caption

3.4.3. Bandpass

There are three bandpass filters in pd, `[bp~]`, `[reson~]` and `[vcf~]`. We can also make our own custom bandpass using a lowpass and a highpass in series:

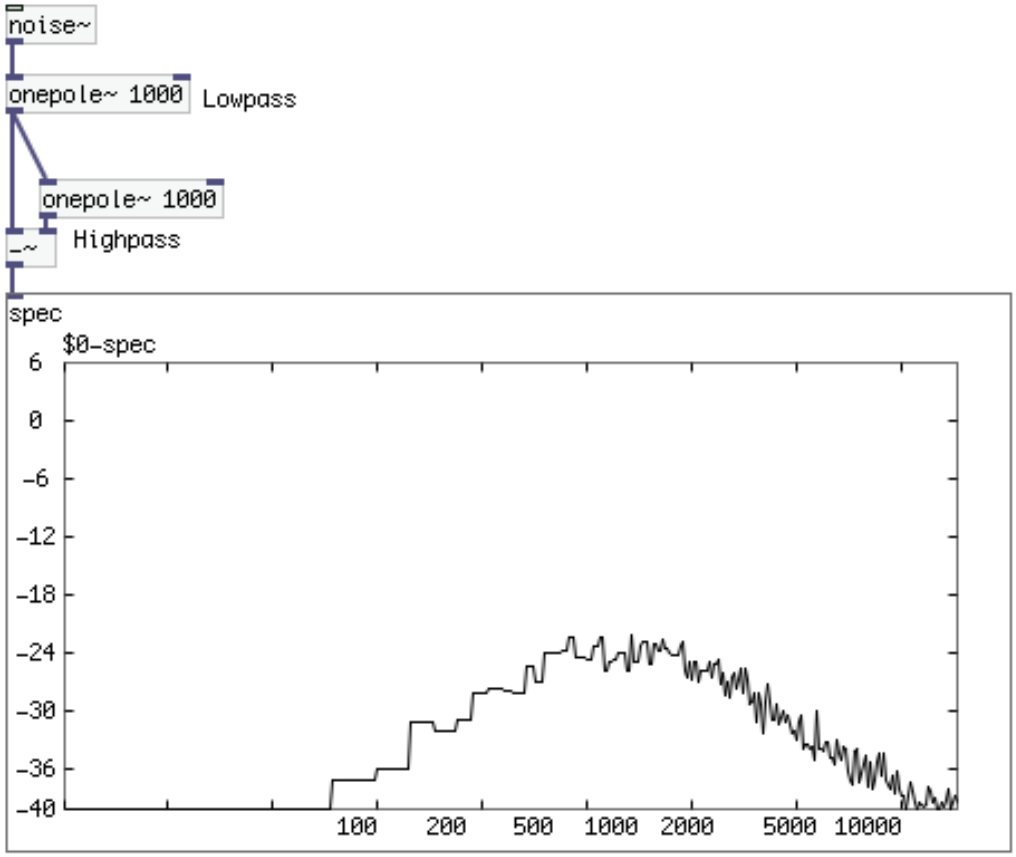


Figure 3.22.: CAPTION MISSING

More basic building blocks that can be used to realize a filter are `[rpole~]`, `[rzero~]`, `[cpole~]`. `[czero~]`. `[biquad~]` is a very powerful filter that allows more or less arbitrary responses.

3.5. Types of Digital Filters

It is important to understand that there are two types of filters[4].

- Finite Impulse Response (FIR) Filters
- Infinite Impulse Response (IIR) Filters

[4]We are going to ignore that one can also filter by using FFT approaches

We have been dealing with a FIR filter so far. An infinite impulse response filter does not have an infinite impulse response in practice, but it *can* have one. How is this achieved? By using feedback. A FIR filter does not have feedback, an IIR filter does.

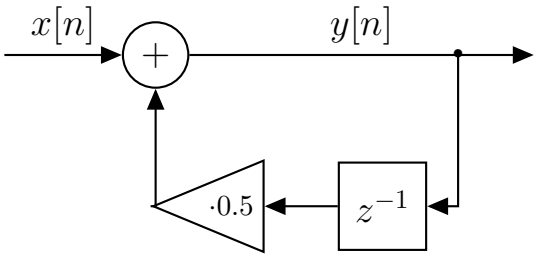


Figure 3.23.: A simple IIR filter. Note that the arrows are circling, there is feedback involved.

If we look at the impulse Response, we can see that it is "longer" than what we had before. If you look closely, we can see that the output sample at which the impulse arrives(n=3), just has the value of the input(1). After that, we can see that the next value is always the sample before times 0.5, so the impulse respopnse is a *geometric series*: $\{1, 0.5, 0.25, 0.125, ..\}$ or $\{1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, ...\}$. This is the case since the coefficient in the feedback path is 0.5, so $\frac{1}{2}$.

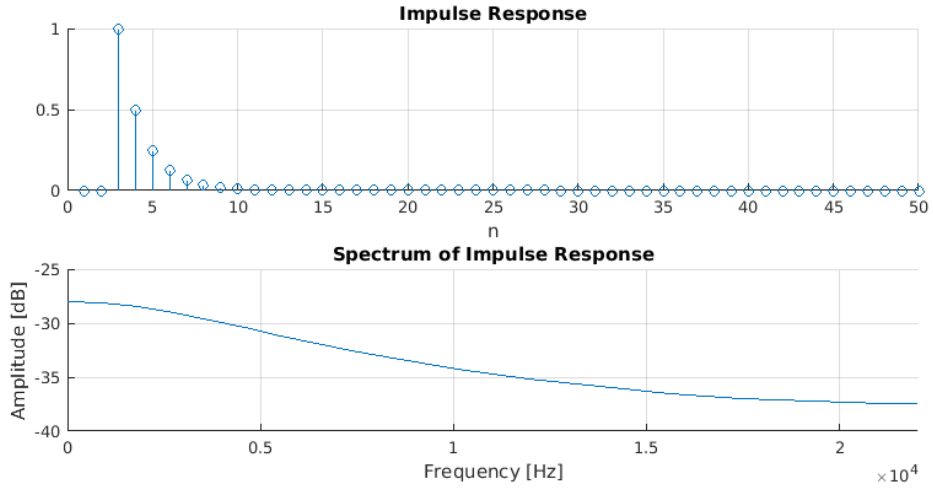


Figure 3.24.: CAPTION MISSING

Let's translate the above diagram to a difference equation, so we see another representation of the same system.

$$y(n) = x(n) + y(n-1) \cdot 0.5 \tag{3.10}$$

Finally, let's quickly summarize some of the differences between the two filter types:

	IIR	**FIR**
stability	possibly unstable	always stable
Real time Efficiency	Very efficient	inefficient
topology	Feedback	no feedback

3.6. Key Points

- Make sure you understood the differences between a FIR and an IIR filter.
- Make sure you can make an educated guess what kind of a filter we see if you encounter a block diagram.
- Make sure you can do simple translations between pd, blockdiagrams and difference equations.
- Make sure you know when a combfilter turns into a lowpass.
- Make sure you know and understand the formula to calculate the first phase cancellation in a combfilter.

4. Fourier Transformation

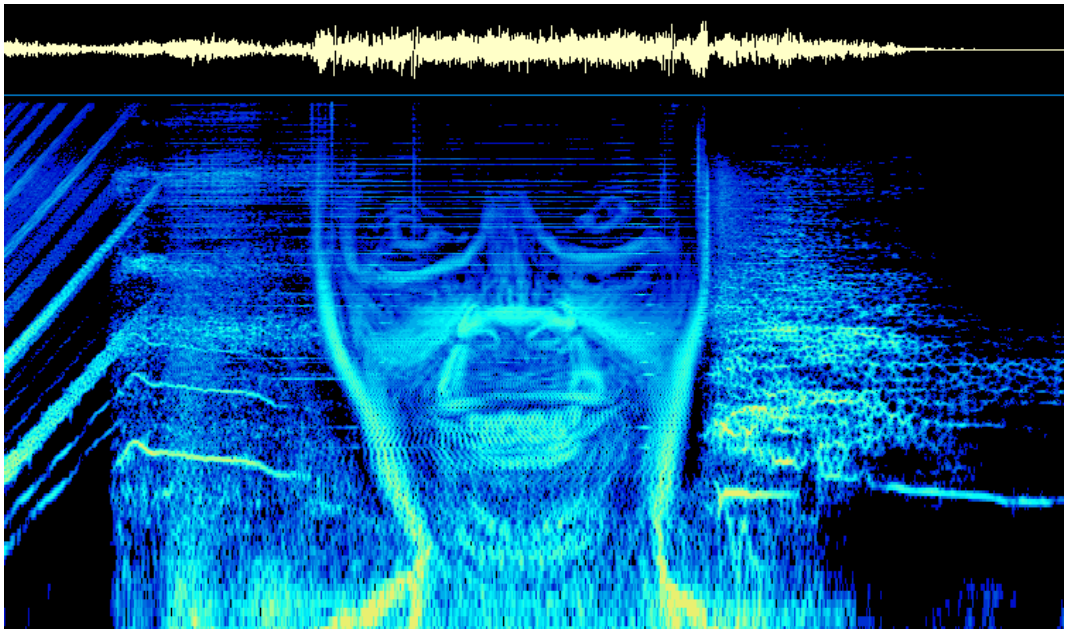


Figure 4.1.: Image in the spectrogram in a track by “Aphex Twin”.

4.1. Notizen

vorbereitung:
Eulers identity, complex numbers.
Initialisierung Fourier transformation. Spectral filter Spectral Reverb, delay. Windowing Convolution, evtl. cross correlation freq. crossover spectral synyth, spectraum display.
evtl auch:

- send / receive, send / receive bei gui objekten.
- initialisierung
-

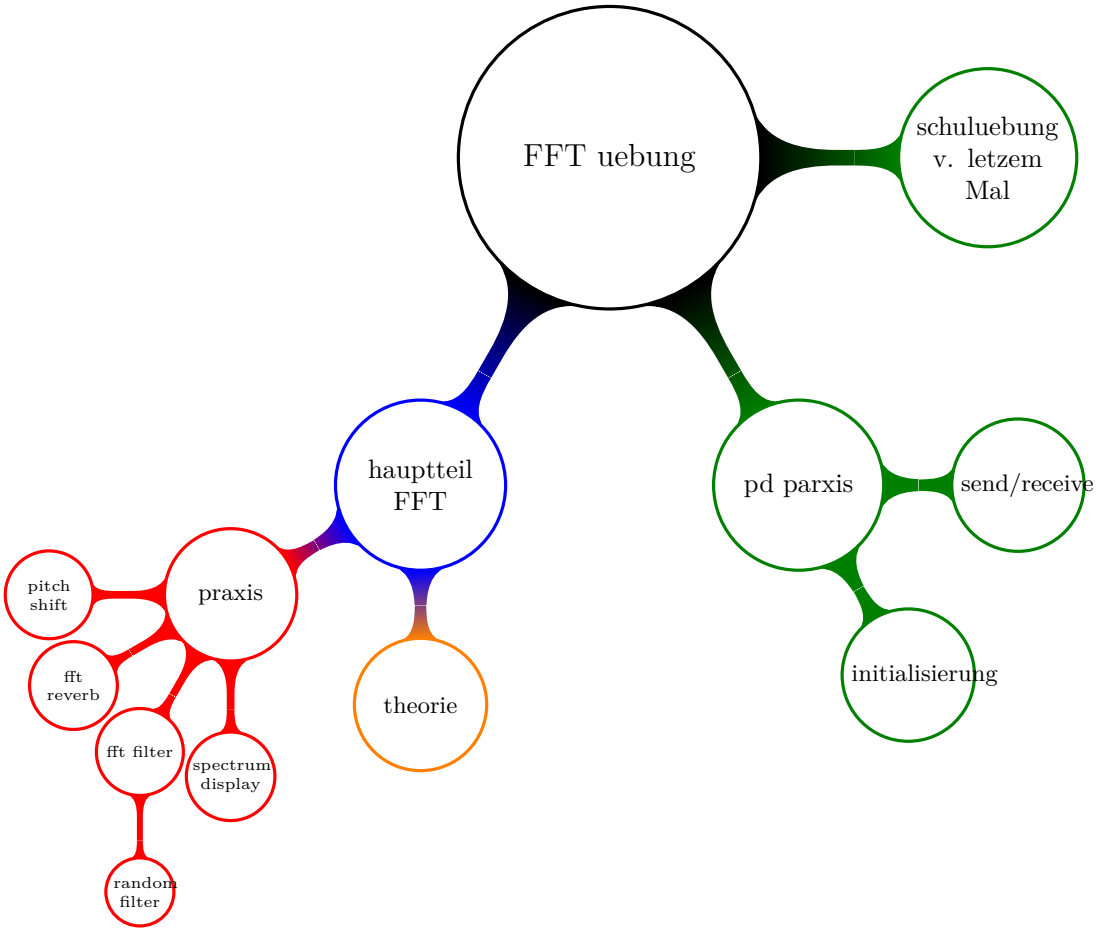


Figure 4.2.: Lecture Contents: Fourier Transformation

Diskretes Signal -> Periodisches Spectrum
Periodisches Signal -> Diskretes Spectrum
gute referenz: http://jackschaedler.github.io/circles-sines-signals

4.2. Fourier Transformation

Geschichte: Bernoulli, Euler, Gauß, Fourier
Eulers identität:

$$e^{ix} = cos(x) + i \cdot sin(x) \tag{4.1}$$

Fourier Transformation:

$$X(f) = \mathcal{F}\{x(t)\} = \int_{-\infty}^{\infty} x(t) e^{-j2\pi ft}\,\mathrm{d}t \tag{4.2}$$

Inverse Fourier Transformation:

$$x(t) = \mathcal{F}^{-1}\{X(f)\} = \int_{-\infty}^{\infty} X(f) e^{j2\pi ft}\,\mathrm{d}f \tag{4.3}$$

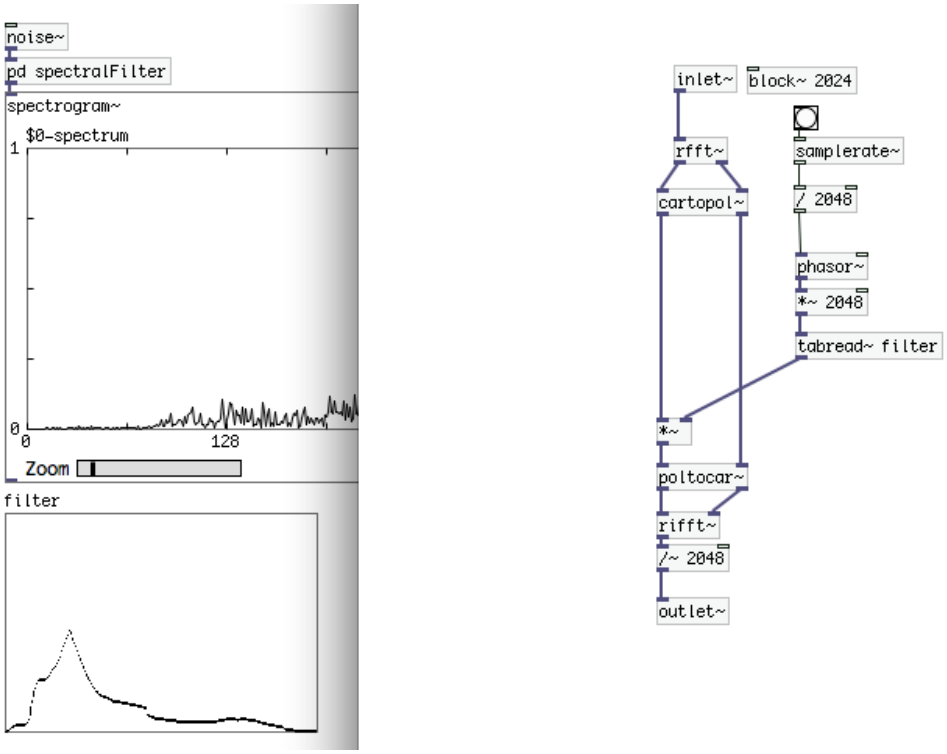


Figure 4.3.: spectralFilter.pd

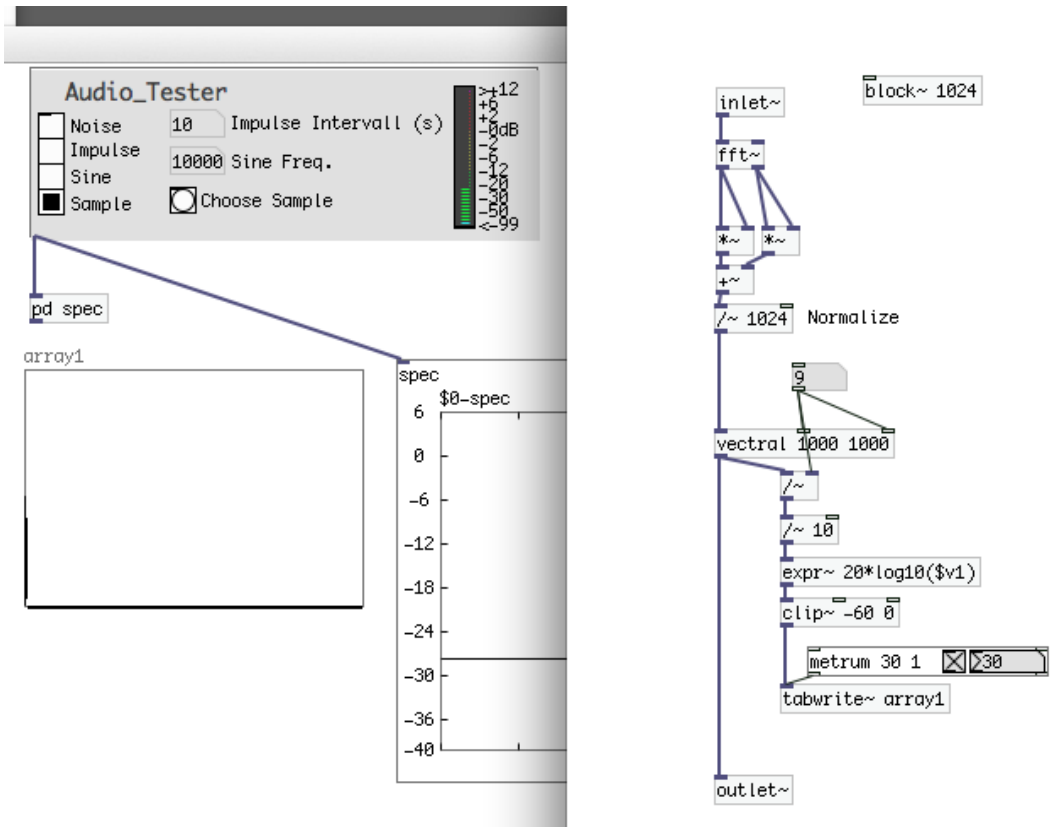


Figure 4.4.: showspectrum

List of Figures

1.1. shortCaption II
1.2. simple sine plot IV
1.3. shortCaption V
1.4. shortCaption V
1.5. shortCaption VI
1.6. Spacial aliasing in graphics. The "frequency" of the intended pixels is to high for the actual pixels. VII
1.7. shortCaption VIII
1.8. Aliasing in the Frequency Domain IX
1.9. a sine wave X
1.10. a sine wave X
1.11. sine 0 to 1 XI
1.12. shortCaption XII
1.13. Dc-offset in pd XII
1.14. The dirac object XIII
1.15. caption XIV
1.16. message domain vs. signal domain XVII
1.17. shortCaption XVIII
1.18. shortCaption XIX
1.19. shortCaption XX

1.1. Lecture Contents 2
1.2. Wave shaped sine oscillator 3
1.3. Distorted sine, time domain 3
1.4. Linear Transfer function 4
1.5. shortCaption 5
1.6. farnell wave shaping visualization 5
1.7. The function $f(x) = x^2$ 6
1.8. Contrast curve in photoshop 7
1.9. The square function in pure data, using a 100Hz sine as a test signal. What do you hear? 7
1.10. Applying the square function to an input sine wave. 8
1.11. Left: using a mathematical function to calculate the output. Right: using a table to look up the output. 10

1.12. farnell waveshaping identity 11
1.13. clipped sine wave 12
1.14. clipped sine wave 2 13
1.15. Clipping an amplified sine wave in pd 13
1.16. An amplified and clipped sine wave in the time domain. 14
1.17. simpleSampler 15
1.18. sound in Ram 16
1.19. RamFilePlayback 17
1.20. writing Audio to disk 18
1.21. moreSamplimg.pd, a simplified version of granular sampling 19
1.22. audioTester.pd, zu bauen als Hausübung 20

2.1. “Surface Modulation” by Richard Sweeney 21
2.2. Lecture Contents 22
2.3. Simplest form of “Ring modulation”. 23
2.4. AM time domain 24
2.5. adding two oscillators 25
2.6. caption 26
2.7. The General Idea of FM 27
2.8. Naive Implementation with Direct Parametrization. 28
2.9. FM with Index and Ratio 30
2.10. shortCaption 31
2.11. shortCaption 31
2.12. shortCaption 32

3.1. Caravaggio, Narziss. ad Feedback: Echo liebt Narziss etc. 33
3.2. Original Signal 34
3.3. Filtered Signal 1 35
3.4. Filtered Signal 2 36
3.5. shortCaption 37
3.6. A very simple lowpass filter. 38
3.7. shortCaption 40
3.8. shortCaption 41
3.9. FIR filter in pd 41
3.10. shortCaption 43
3.11. Destructive Interference 43
3.12. shortCaption 45
3.13. Spectrum Impulse, Lowpass, DC 46
3.14. shortCaption 47
3.15. A very simple highpass filter. Please compare to 3.6. 48
3.16. Video Filters: Original 49

3.17. Video Filters: Blur 50
3.18. Video Filters: IIR Lowpass, time domain 51
3.19. Video Filters: FIR Highpass, space domain 52
3.20. Video Filters: FIR Highpass, time domain 52
3.21. caption 53
3.22. bandpass 54
3.23. A simple IIR filter. Note that the arrows are circling, there is feedback involved. 55
3.24. shortCaption 55

4.1. Image in the spectrogram in a track by “Aphex Twin”. 57
4.2. Lecture Contents: Fourier Transformation 58
4.3. spectralFilter.pd 59
4.4. showspectrum 60

Bibliography

Farnell, A. (2010). *Designing sound*. MIT Press, Cambridge, Mass.

Miller Puckette (2006). The Theory and Technique of Electronic Music.

Smith, J. O. (2007). Introduction to Digital Filters with Audio Applications.