Class 10:
Spectrum and Filters

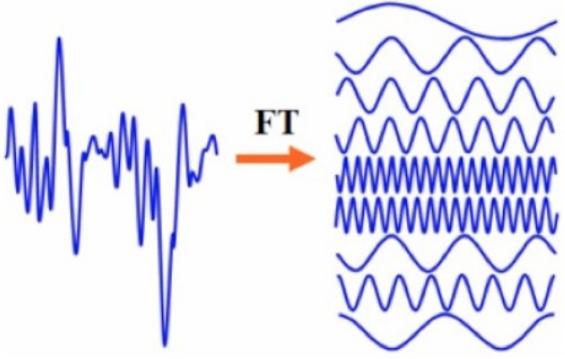
February 20, 2020

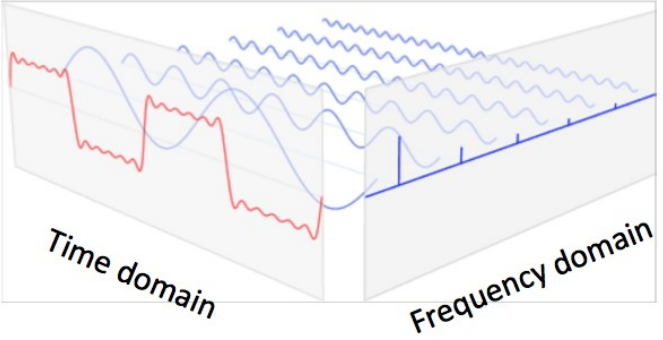
MUSC 3540: Embedded Audio Programming

Fourier Theory

Jean-Baptiste Joseph Fourier
(1768-1830)

- Any wave can be broken down into individual sine wave components → partials
- This is just an approximation of the wave, but good!
- Enables analysis and resynthesis using partials

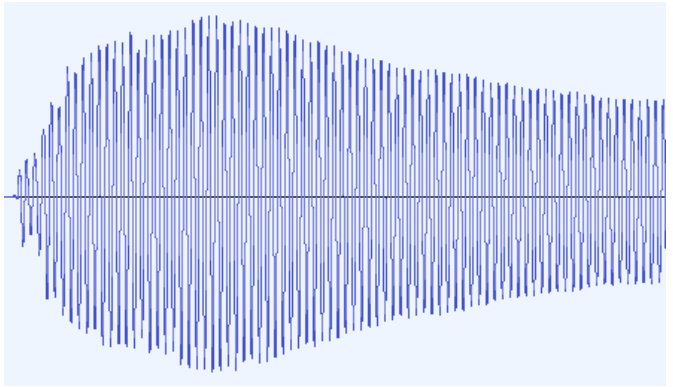


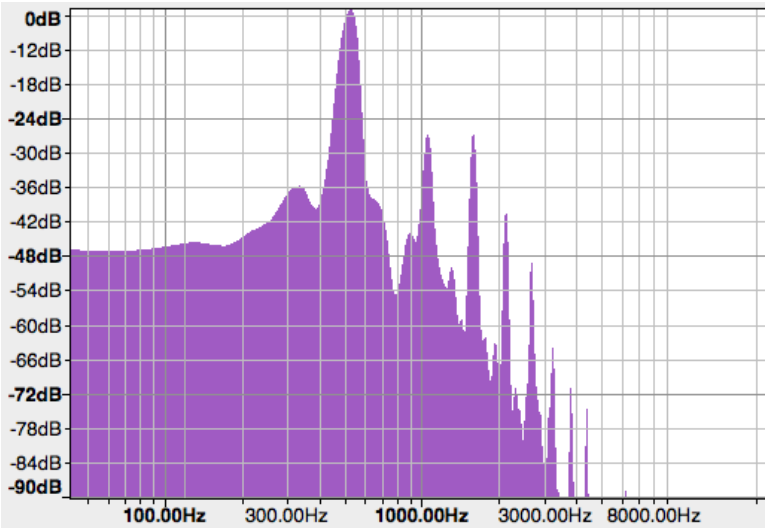


- When a wave is the result of the combination of other waves, we call it *waveform*

Spectrum

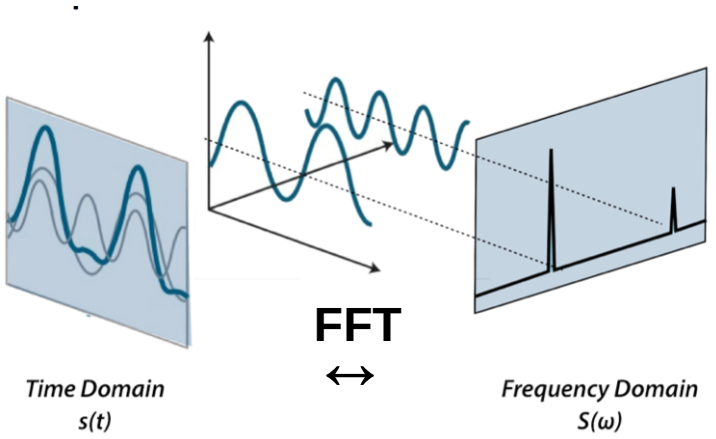
- We can represent a wave by means of showing its partials, rather than the resulting waveform!
- If we represent the amplitude of the partials as a function of frequency, we generate the *spectrum* of the wave!
  - “spectrum” means “variety” [of frequencies]





Fast Fourier Transform

- Algorithm that allows to extract spectrum of digital representation of wave!
- Bidirectional: Time ↔ Frequency
- The Fast Fourier Transform (FFT)and the inverse FFT allow for the conversion of any digital signal (series of samples) to the frequency domain and back again to either the time.



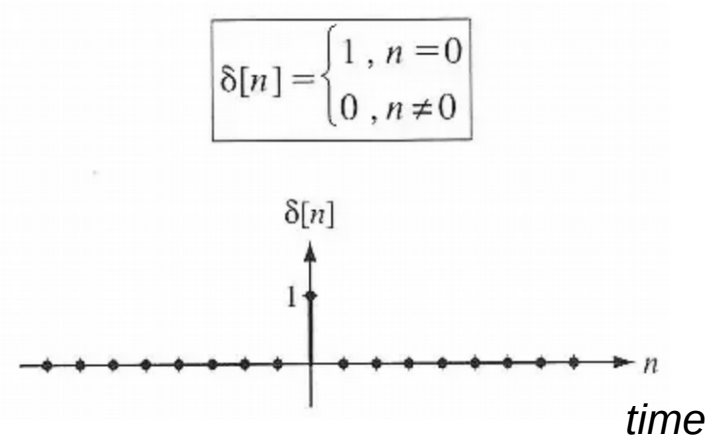
Bela Scope FFT

- The oscilloscope built in Bela allows us to quickly compute the FFT of up to 4 signals and visualize the resulting magnitude spectra
- Let’s create a new C++ project and upload the render file that you find in the “osc_spectra” within today’s class zip folder.
- Then, let’s add these lines:

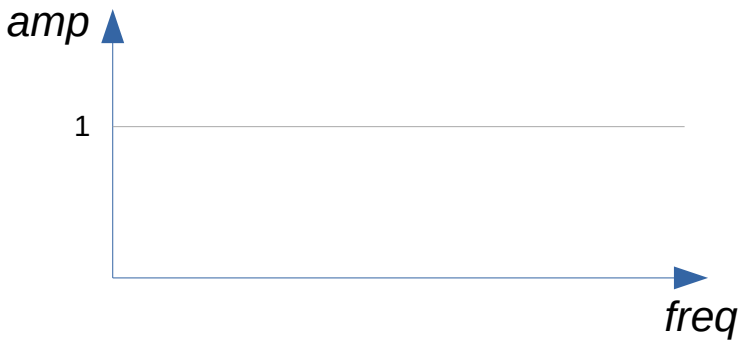
```
#include "libraries/Scope/Scope.h" // add Scope lib

Scope scope; // declare Scope object

// in setup()
scope.setup(2, context->audioSampleRate); // prepare 2 channels on scope

// in render()
scope.log(sample_sine, sample_square); // send samples to scope
```

The Strange Case of the Impulse

- Impulse: a single instantaneous pulse
  - max amplitude (let's assume = 1)
  - but infinitesimal duration

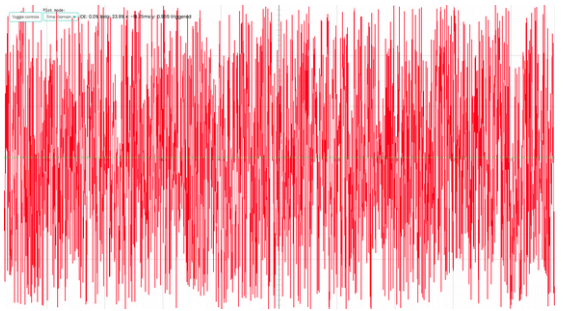


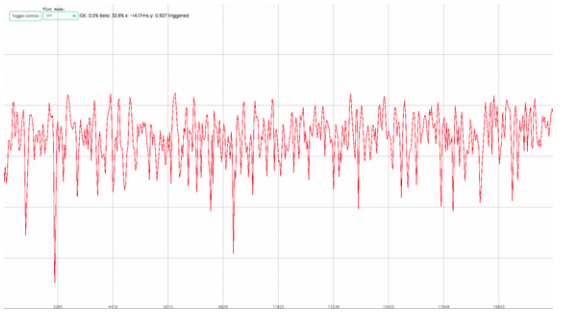
- The spectrum of the impulse is a flat line!
  - All frequencies, same amplitude (= 1)



Noise

- Aperiodic wave based on randomness:
  - no repeating pattern in time
  - stochastic distribution of frequencies
- White noise: random across all spectrum
  - Flat distribution across whole frequency range → very similar to impulse
  - Difficult to find in nature, but useful for experiments





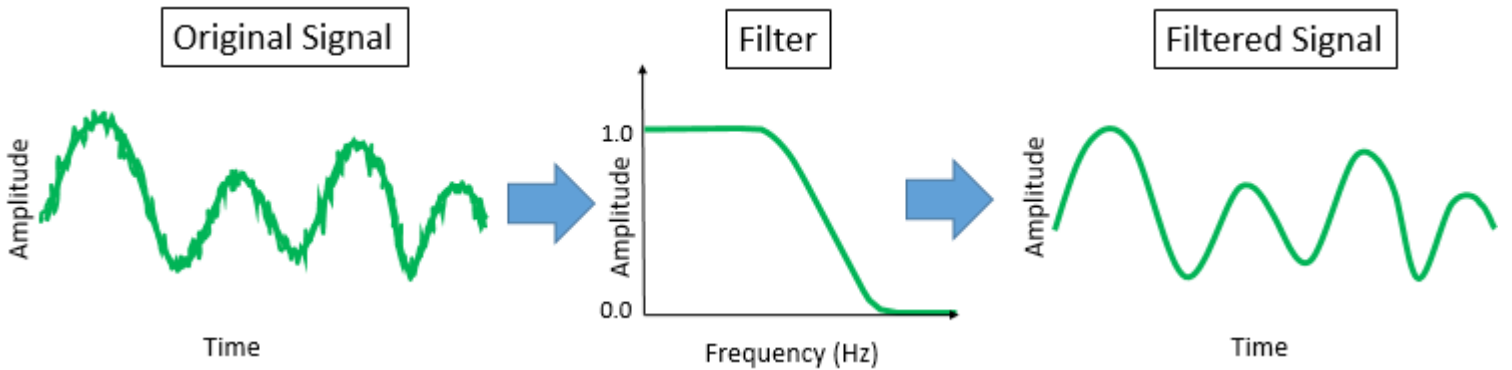
White Noise in C++

- Create a new C++ project and upload the render file from project “white_noise”

- The standard C++ function *rand()* is the key to “simulate” white noise!
- “Simulate”, because true randomness cannot be achieved on computers
- *rand()* outputs a semi-random series of numbers, which is good enough for us!

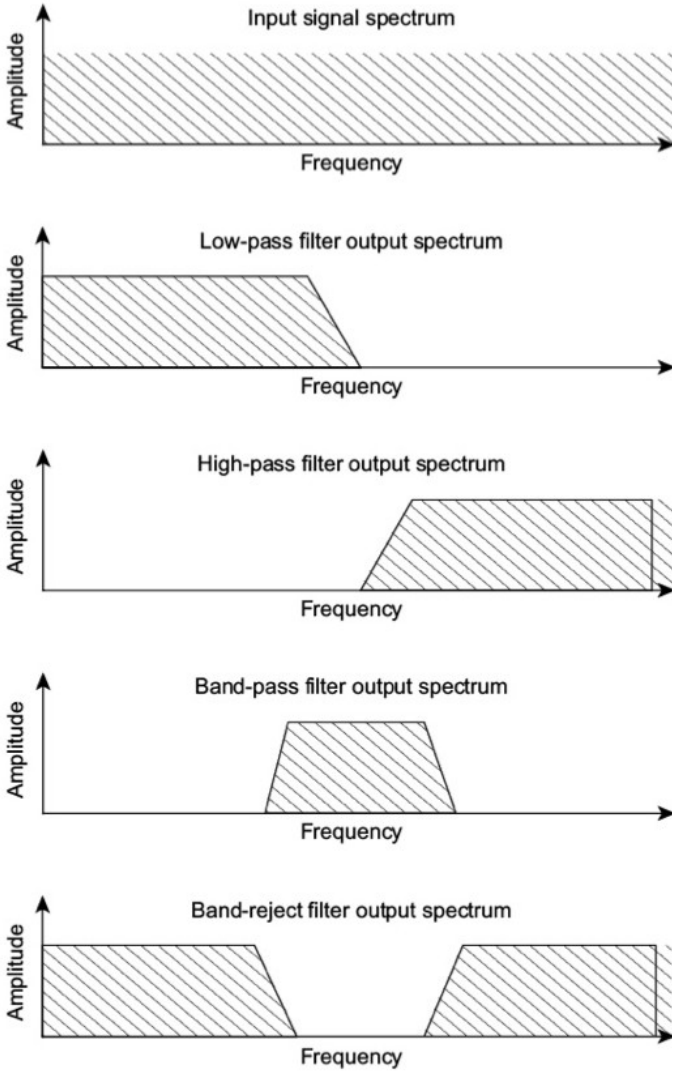
Filters

- A filter is a system that modifies the spectrum of an input signal
- As a consequence it outputs a time signal that is related to the input, but different



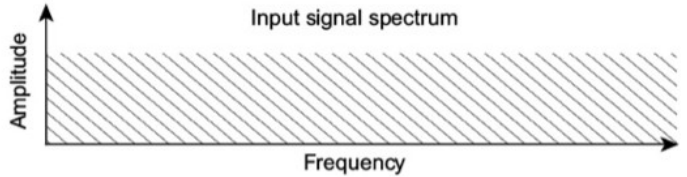
Basic Types of Filters

- Four basic types:
  - Low-pass
  - High-pass
  - Band-pass
  - Band-reject
- Change the input spectrum differently!

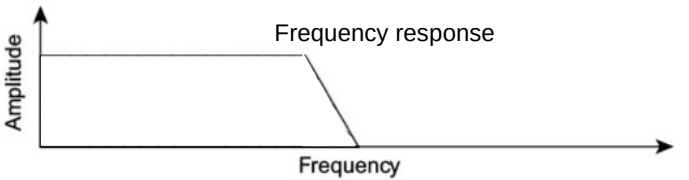


Frequency Response

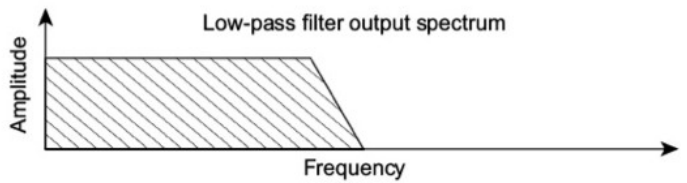
- The frequency response of a filter shows how the filter modifies the input spectrum
- Output spectrum =
  signal spectrum *x* filter frequency response
- Frequency response computed as follows:
  - send impulse into filter (impulse response)
  - FFT of impulse response = freq response

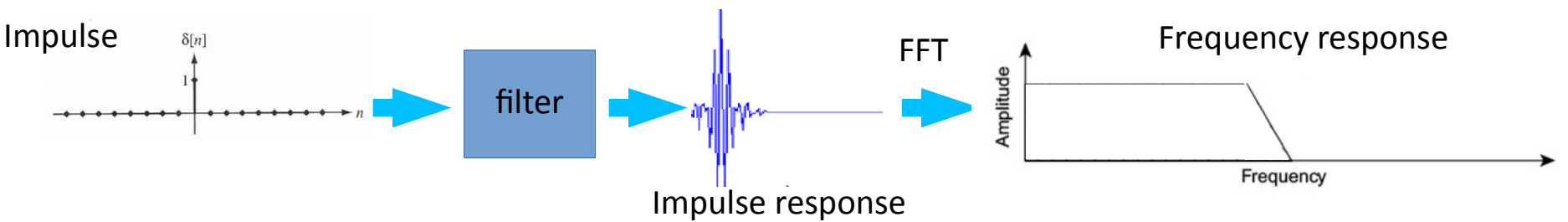


x



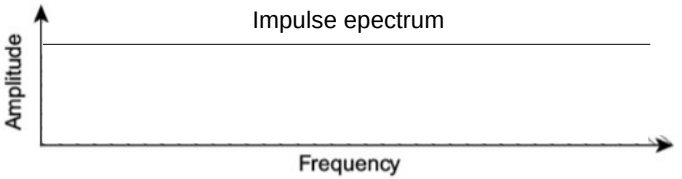
=

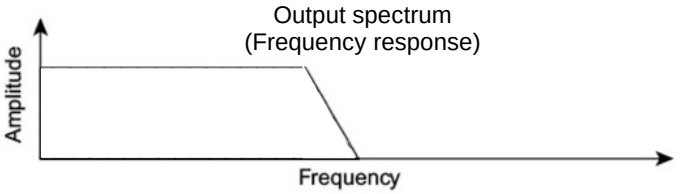


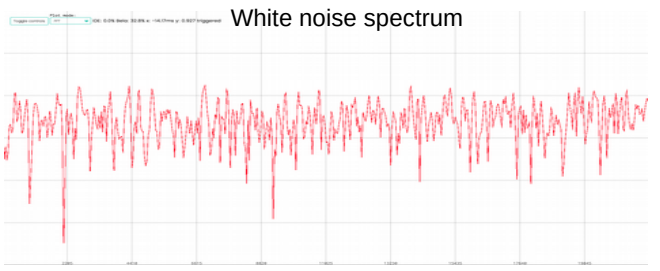


Frequency Response

- Impulse because it contains all frequencies
  - Most complete input spectrum!
  - The result shows the effects of the filter on all frequencies
- We can use white noise too!
  - Over time, it contains all frequencies
  - And it lasts longer
- We can continuously visualize a good approximation of the frequency response







Digital Filters

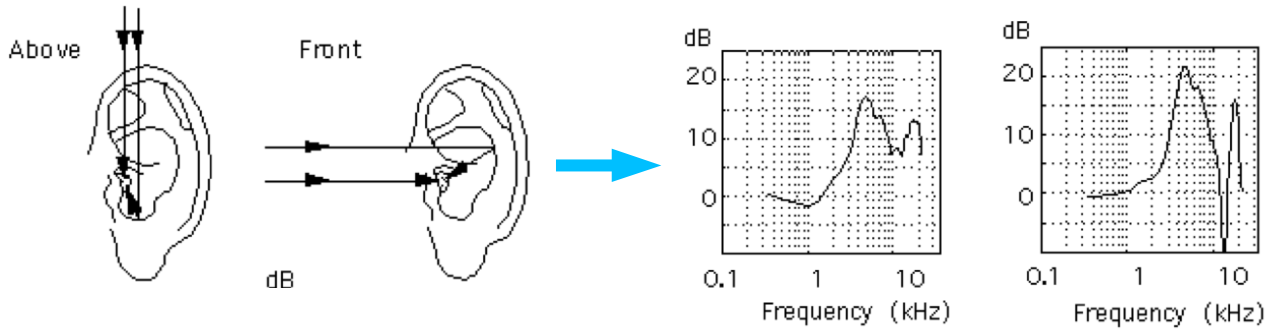
- In digital signal processing, the simplest filters are implemented as a weighted sum of the previous inputs (including the current one)
- These are called Finite Impulse Response (FIR) filters and can be generically represented as:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

- y[n] = current output
- x[n] = current input [sample], x[n-1] = previous input, x[n-2] = two inputs ago, etc.
- b_k = weights of the inputs

Biomechanical Filter

- In other words, in this case filtering is obtained by adding together multiple delayed copies of a signal/wave
- This is very similar to what happens to a sound when it is reflected by our outer ear and gets into the aural canal
- The biomehcanical filtering effect is used to estimate the elevation of the sound source



FIR Filters

- M is the *order* of the filter, i.e., time index of the oldest input that we use

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

- By choosing different weights b_k, we can change the behavior of the filter (e.g., type, cut-off frequency)
- In some filters, some b_k can be equal to zero!

First Order FIR in C++

- The implementation of FIR filters in C++ is quite straightforward, for we can easily save the previous inputs *x[n-k]*!

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

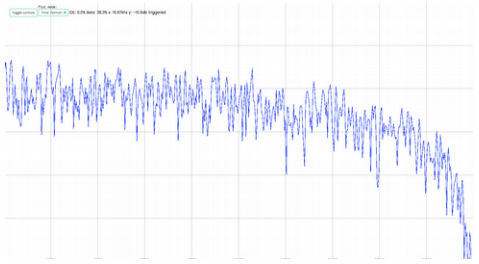
- Let's implement a first order FIR (M=1) → it uses only the current and the previous inputs:

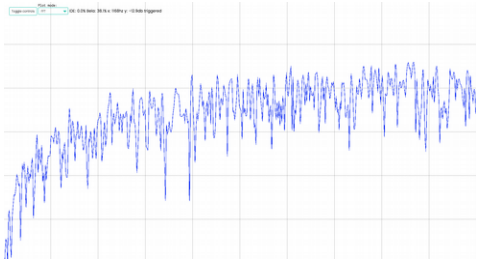
$$y[n] = b_0 x[n] + b_1 x[n-1]$$

- Create a new project and copy the render file from "fir"
  - Let's first try: $b_0 = 1$, $b_1 = 1$
  - Then: $b_0 = 1$, $b_1 = -1$

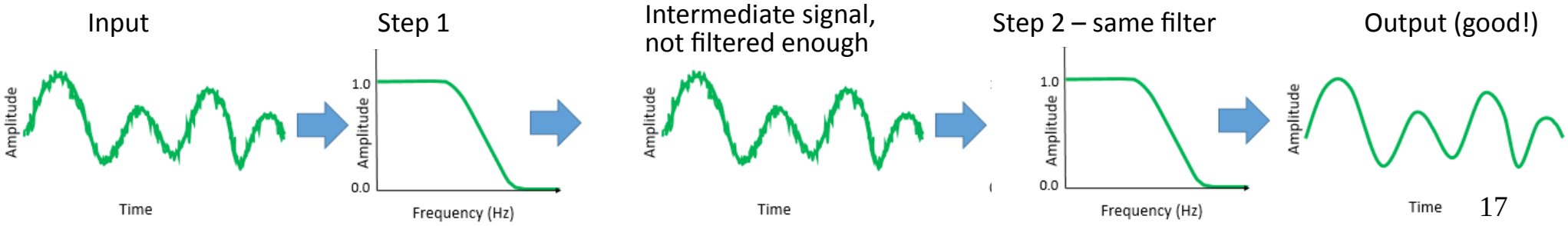
Two-step Filter

- Not very aggressive cut-off frequencies! Gentle slopes…



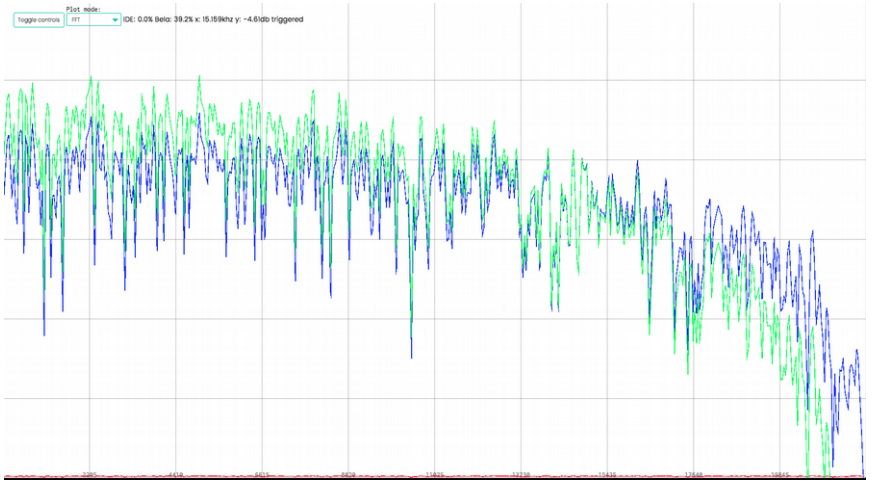


- We can get better results by putting two filters in cascade → two step filtering!
- Create a new project and copy the render file from “fir_2step”



More Steps...

- As you can see the cut-off frequency is more precise, but the difference is minimal



- Let’s see two ways to get better cut-offs!

IIR Filters

- Infinite Impulse Response (IIR) filters are slightly more complex than FIR ones, but they allow for more precise cut-off frequencies
- They combine both previous inputs and previous outputs:

$$y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$

- The order is *max(N, M)*
- More coefficients, hence more operations → computationally heavier compared to FIR of same order
  - Output summation starts from 1! → because $a_0 = 1$, always!

First Order IIR

- Let's implement a very simple first order IIR filter, with M=N=1:

$$y[n] = b_0 x[n] + b_1 x[n-1] + a_1 y[n-1]$$

- This time, let's see the equations to compute the weights according to the desired cut-off frequency:

Low pass

$$a_1 = e^{-2\pi \frac{f_c}{f_s}}$$
$$b_0 = 1 - e^{-2\pi \frac{f_c}{f_s}}$$
$$b_1 = 0$$

High pass

$$a_1 = e^{-2\pi \frac{f_c}{f_s}}$$
$$b_0 = \frac{1 + e^{-2\pi \frac{f_c}{f_s}}}{2}$$
$$b_1 = -\frac{1 + e^{-2\pi \frac{f_c}{f_s}}}{2}$$

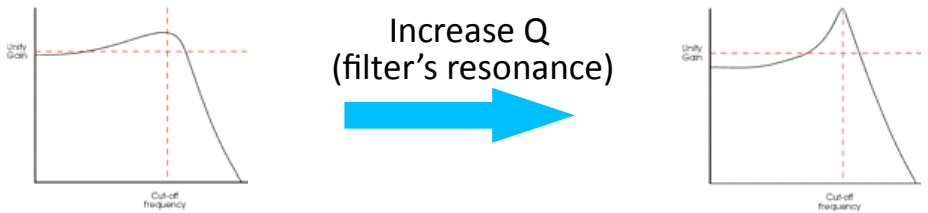
fc = cut-off freq

fs = sample rate

- Create a new project and copy the render file from "iir"

Higher Order Filters

- Both IIRs and FIRs are more effective when of higher order
  - More coefficients, higher precision and more parameters available (e.g., resonance)

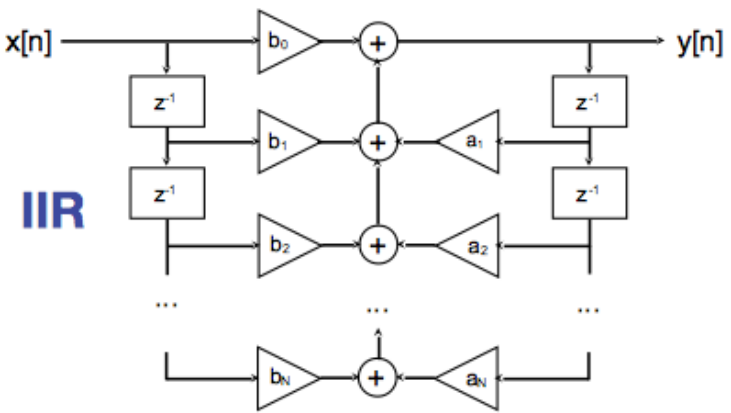


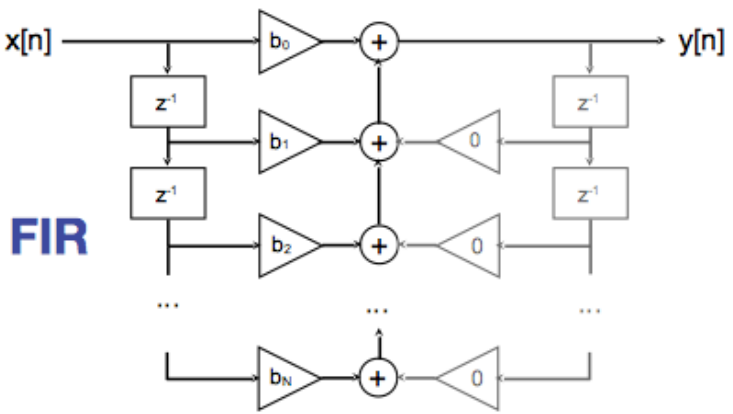
- But more coefficients means:
  - more previous inputs/outputs to save → more memory needed
  - more calculations to filter the signal → more computational power needed
- And both higher order FIRs and higher order IIRs can have more steps, to further improve precision, at the expenses of resources usage

Direct Form 1

- The generic equation: $y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$

  is the implementation of the Direct Form 1 of IIRs and FIRs:

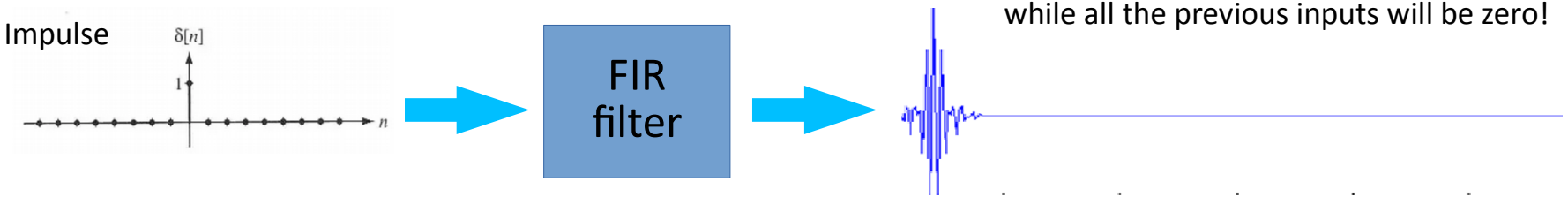




- z^{-1} is a unit delay block → delays of one sample (input or output)

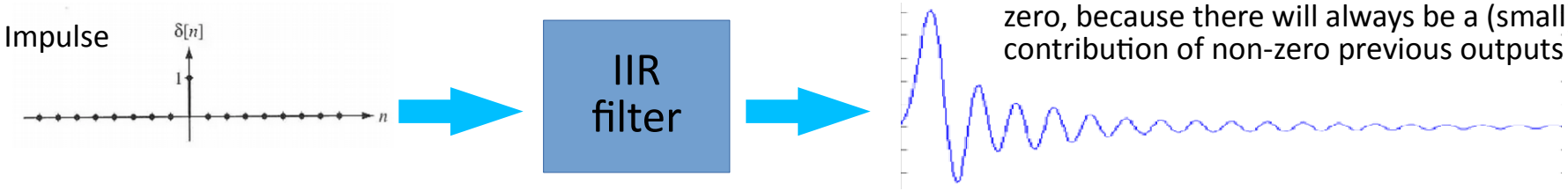
Impulse Response Duration

- Finite Impulse Response:



The response goes to zero, because after a while all the previous inputs will be zero!

- Infinite Impulse Response:



The response will asymptotically tend to zero, because there will always be a (small) contribution of non-zero previous outputs