SON 2026-2027 lecture 6

3TC Audio Project @ INSA Lyon

Tanguy Risset & Romain Michon

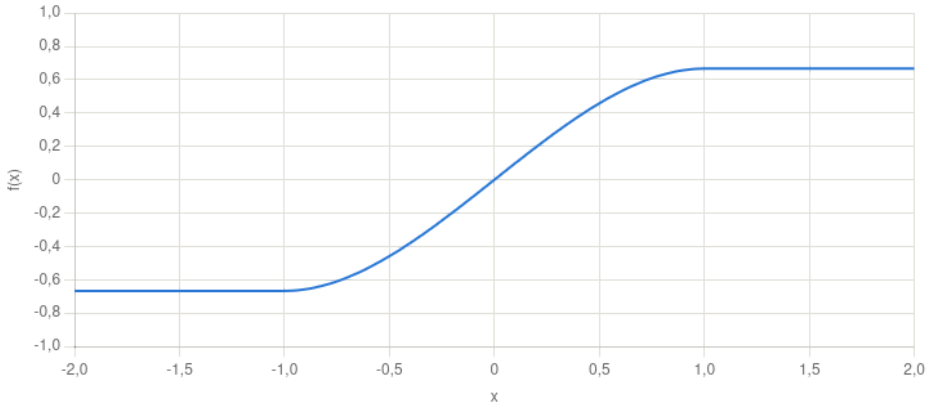
https://inria-emeraude.github.io/son/

Audio Processing Basics II

Harmonic Distortion

- Distortion is one of the most common electric guitar effect.
- It consists of over driving a signal by increasing its gain to "square" the extremities of its waveform.
- Lots of harmonics, producing very "rich" sounds.
- Overdrive is easily achievable with an analog electronic circuit and "sharp edges" in the waveform are rounded thanks to the tolerance of the electronic components.
- In the digital world, things are slightly more complicated since clipping will happen resulting in a very dirty sound with potentially lots of aliasing.
- One way to solve this problem is to use a "cubic function" which will round the edges of the signal above a certain amplitude:

$$f(x) = \begin{cases} \frac{-2}{3}, & x \leq -1 \\ x - \frac{x^3}{3}, & -1 < x < 1 \\ \frac{2}{3}, & x \geq 1 \end{cases}$$



The cubic function

Harmonic Distortion in MyDsp

- `Distortion.cpp` implements a cubic.
- The range of `drive` is {0;1} which means that the value of `input` can be multiplied by a number as great as 100 here.
- `offset` is a common parameter which just adds a positive or negative DC offset to the signal.
- If this parameter is used, it is recommended to add a DC blocking filter after the distortion.

Distortion is created here by clipping the signal using the `fmin` and `fmax` functions. Finally, the cubic polynomial is used to round the edges of the waveform of the signal as explained above.

The `distortion` example program for the Teensy demonstrates the use of `Distortion.cpp`.

```
float Distortion::cubic(float x){
  return x - x*x*x/3;
}

float Distortion::tick(float input){
  float output = input*pow(10.0,2*drive) + offset;
  output = fmax(-1,fmin(1,output));
  output = cubic(output);
  return output*gain;
}
```

Echo

- An echo is a very common audio effect which is used a lot to add some density and depth to a sound.
- It is based on a feedback loop and a delay and can be expressed as:

$$y(n) = x(n) + g.y(n - M)$$

- where g is the feedback between 0 and 1 and M the delay as a number of samples.
- Here, `delBuffer` is used as a "ring buffer":
  - incoming samples are stored at the `writeIndex`
  - r `writeIndex` and `readIndex` are used to write incoming samples and read previous ones.

- `Echo.cpp` implements an echo as:

```
[...]
delBuffer = new float[maxDel];
[...]
float Echo::tick(float input){
  float output = input + delBuffer[readIndex]*feedback;
  delBuffer[writeIndex] = output;
  readIndex = (readIndex+1)%del;
  writeIndex = (writeIndex+1)%del;
  return output;
}
```

- Note that memory is allocated in the constructor of the class for `delBuffer` based on the value of `maxDel`, the maximum size of the delay.
- The `echo` example program for the Teensy demonstrates the use of `Echo.cpp`.

Comb

- A comb filter is a filter whose frequency response looks like a "comb."
- Comb filters can be implemented with feed-forward filters (Finite Impulse Response -- FIR) or feedback filters (Infinite Impulse Response -- IIR).
- In fact, the Echo algorithm can be used as a comb filter if the delay is very short:

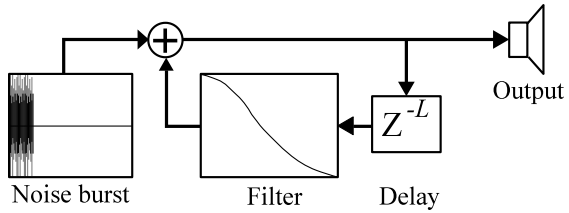
$$y(n) = x(n) - g.y(n - M)$$

- where M is the length of the delay and g feedback coefficient.
- Julius Smith's website presents the frequency response of such filter and the mathematical rationals behind it.
- A feedback comb filter will introduce resonances at specific point in the spectrum of the sound.
- The position and the spacing of these resonances is determined by the value of M.
- g will determine the amplitude and sharpness of these resonances.
- The `comb` example program for the Teensy demonstrates the use of `Echo.cpp` as a comb filter.

Physical Modeling: the Simple Case of the Karplus Strong

- Physical modeling is one of the most advanced sound synthesis technique and a very active field of research.
- It consists of using physics/mathematical models of musical instruments or vibrating structures to synthesize sound.
- Various physical modeling techniques are used in the field of audio synthesis:
  - Mass/Interaction (MI),
  - Finite Difference Scheme (FDS),
  - Signal models (e.g., waveguides, modal systems, etc.).
- An extremely primitive string model can be implemented using a delay line and a loop.
- This primitive string model is called the *"Karplus-Strong"* algorithm

- The delay line models the time it takes for vibration in the string to go from one extremity to the other, and the loop models the reflections at the boundaries of the string.



Karplus-Strong Algorithm (Source: Wikipedia)

Karplus Strong Modeling

- The Karplus-Strong algorithm is typically implemented as:

$$y(n) = x(n) + \alpha \frac{y(n-L) + y(n-L-1)}{2}$$

- where:
  - $x(n)$ is the input signal (typically an dirac or a noise burst),
  - α is the feedback coefficient (or dispersion coefficient, in that case),
  - L is the length of the delay and hence, the length of the string.
- $\frac{y(n-L)+y(n-L-1)}{2}$ can be seen as a one zero filter implementing a lowpass. It models the fact that high frequencies are absorbed faster than low frequencies at the extremities of a string.
- The length of the delay L can be controlled as a frequency using the following formula: $L = fs/f$ where f is the desired frequency.
- The system must be excited by a dirac.

Karplus Strong Modeling

`KS.cpp` implements a basic Karplus-Strong algorithm:

```
float KS::tick(){
  float excitation;
  if(trig){
    excitation = 1.0;
    trig = false;
  }
  else{
    excitation = 0.0;
  }
  float output = excitation +
         oneZero(delBuffer[readIndex])*feedback;
  delBuffer[writeIndex] = output;
  readIndex = (readIndex+1)%del;
  writeIndex = (writeIndex+1)%del;
  return output;
}
```

See the Faust Demo of KS

with:

```
float KS::oneZero(float x){
  float output = (x + zeroDel)*0.5;
  zeroDel = output;
  return output;
}
```

The examples folder of the course repository hosts a simple Teensy program illustrating the use of `KS.cpp` .

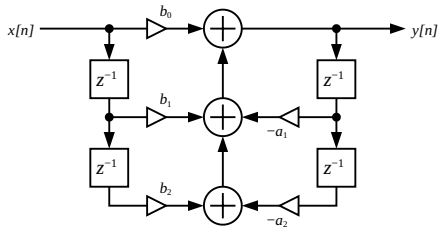
Exercises

- Lowpass, bandpass and highpass filter with biquads
- Peak equalizer
- These two programs are important as they will give you powerfull tools for your SON project.

Biquad Filter: Direct Form 2

- A biquad filter (two poles, two zeros), has the following transfer function:

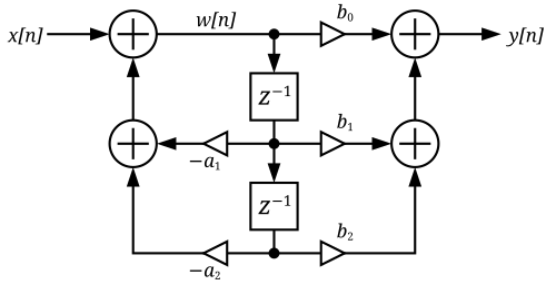
$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$



- *Direct Form 2* is the canonical implementation for embedded systems: a single delay line $w(n)$ is shared between the feedback (poles) and feedforward (zeros) parts, so only 2 state variables (i.e. registers/memory location) are needed instead of 4:

$$w(n) = x(n) - a_1 w(n-1) - a_2 w(n-2)$$

$$y(n) = b_0 w(n) + b_1 w(n-1) + b_2 w(n-2)$$



Direct Form 2 for biquad (Source: Wikipedia)

From Analog Prototypes to digital using the bilinear transform

- Filters are usually designed as an analog (continuous-time) 2nd-order prototype in the s-domain, parameterized by a cutoff/center frequency and a quality factor Q:

$$H(s) = \frac{b_0 + b_1 s + b_2 s^2}{a_0 + a_1 s + s^2}, \quad a_1 = \frac{1}{Q}$$

- Q sets the damping of the pole pair: low Q gives a smooth rolloff, high Q gives a sharp resonant peak.
- Which of b_0, b_1, b_2 are non-zero decides the filter *shape*:
  - lowpass keeps only b_0 which is the gain,
  - bandpass keeps only b_1, etc..
- The analog prototype must be turned into a discrete filter. The *bilinear transform* $s \leftarrow \frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}}$ does this while guaranteeing stability: it maps the whole stable left half of the s-plane inside the unit circle (see Wikipedia bilinear transform).

Implementing biquad in C++

- You first implement the direct form 2 of biquad filter `tf2(b0,b1,b2,a1,a2)` (formula slide 11)
- Then you will then have to format the coefficients of that filter using the bilinear transform and define `tf2s(b2,b1,b0,a1,a0,w1)` (`SR` is the sampling rate):

```
tf2s(b2,b1,b0,a1,a0,w1) = tf2(b0d,b1d,b2d,a1d,a2d)
with {
  c   = 1/tan(w1*0.5/SR);
  csq = c*c;
  d   = a0 + a1 * c + csq;
  b0d = (b0 + b1 * c + b2 * csq)/d;
  b1d = 2 * (b0 - b2 * csq)/d;
  b2d = (b0 - b1 * c + b2 * csq)/d;
  a1d = 2 * (a0 - csq)/d;
  a2d = (a0 - a1*c + csq)/d;
};
```

Making Resonant Lowpass, Bandpass and Highpass

- Finally, you'll have to format the coefficients of the `tf2s` filter depending on the filter you want to realize:
- For the resonant highpass: `cpp resonhp(fc,Q,gain,x) = gain*x-resonlp(fc,Q,gain,x);`

```
resonlp(fc,Q,gain) = tf2s(b2,b1,b0,a1,a0,wc)
with {
     wc = 2*PI*fc;
     a1 = 1/Q;
     a0 = 1;
     b2 = 0;
     b1 = 0;
     b0 = gain;
};
```

(for the resonant lowpass)

```
resonbp(fc,Q,gain) = tf2s(b2,b1,b0,a1,a0,wc)
with {
     wc = 2*PI*fc;
     a1 = 1/Q;
     a0 = 1;
     b2 = 0;
     b1 = gain;
     b0 = 0;
};
```

(for the resonant bandpass)

Exercice: implement the three filter (lowpass, bandpass and highpass)

- Please, note that Q controls the bandwidth of the filter such that: `Q = fc/BW`.
- Wrap this up by plugging a broadband signal generator (e.g., sawtooth oscillator or white noise generator) to the filter.
- Come up with some nice mapping controlled with hardware sensors (i.e., rotary pot, etc.).
- Test your filter by changing dynamically the cutoff frequency of the filter using a potentiometer, for example.

Peak Equalizers

- Peak equalizers are yet another kind of filters allowing to reduce or increase some bands in the spectrum of a sound.
- A peak equalizer can be implemented as this (`tf2s` can be found above) ->
  - `Lfx` controls the level of the filter in dB (0 for no filtering, negative value for band reduction, and positive value for band amplification).
  - `fx` is the center frequency,
  - `B` the bandwidth in Hz.
- Implement this filter and test it the same way as you did for the resonant lowpass/bandpass/highpass.

```
peak_eq(Lfx,fx,B) = tf2s(1,b1s,1,a1s,1,wx) with {
  T = 1.0/SR;
  Bw = B*T/sin(wx*T); // prewarp s-bandwidth
                      //for more accuracy in z-plane
  a1 = PI*Bw;
  b1 = g*a1;
  g = db2linear(abs(Lfx));
  if(Lfx>0) {
    b1s = b1;
    a1s = a1;
  }
  else {
    b1s = a1;
    a1s = b1;
  }
  wx = 2*PI*fx;
};
```

End of Lecture 6