Politecnico di Milano
Computer Music: Languages and Systems
Academic Year 2022-2023
Eray Özgünay – Sebastian Mendez – Harry Foley

Homework 1: Supercollider didactic Tool for Distortion Effects

Introduction

The project brief is to implement a didactic tool in Supercollider to learn and/or teach the audio effect of Distortion. We decided to design an application in Supercollider that could be used by both Student and Teacher.

System Overview

The application features two different modes: 'Learn' and 'Teach', the user selects the mode based on what they want to use the application for.

Learn Mode

Learn Mode is designed to be used by a novice student who has a little to no understanding of what Distortion is or how it works. The purpose of this mode is to introduce the student to the basic principles with a limited access of control to avoid the student being overwhelmed.

The user can select from three different preset audio effects and learn intuitively the differences between them, through the visualisation of the transfer function, the effect on the audio signal in the oscilloscope and of course aurally.

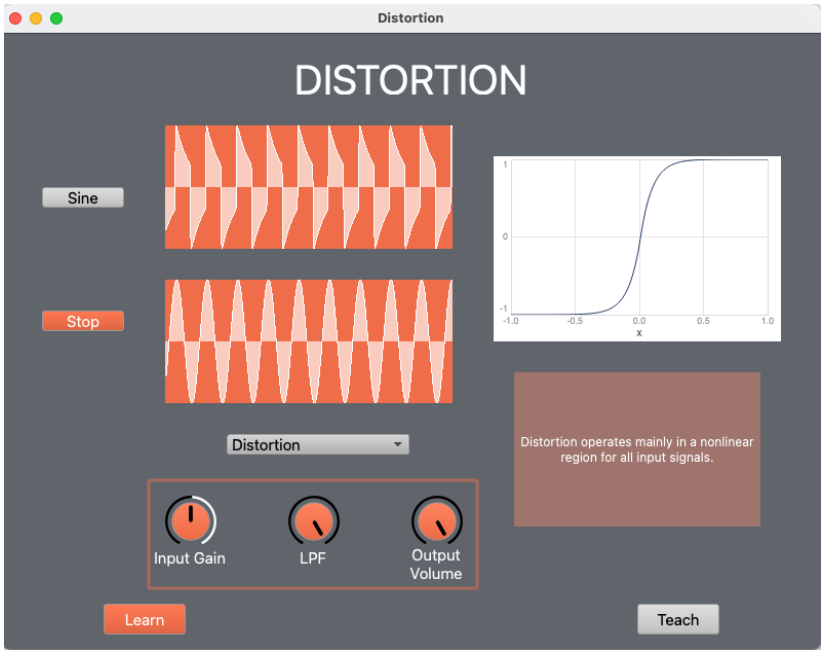

Figure 1. Learn Mode GUI

User Controls:

- Audio Source
  - Sinewave
  - Wav File
- Pre-set:
  - Overdrive
  - Distortion
  - Fuzz
- Input Gain
- LPF Cut-off Frequency
- Output Volume

Teach Mode

Teach Mode is designed to be used by a professor who already has a sufficient level of understanding of the distortion effect and how it is implemented. The purpose of this mode is to allow a professor to use this application to teach students how a distortion effect works, its parameters and their effect on an audio signal. For this reason, we give the Professor access to all the distortion effect's parameters and remove the use of presets. With an extensive control of the effect and views of the Oscilloscope, Frequency Scope and Transfer Function we believe the professor can demonstrate a wide range of information clearly to a group of students.

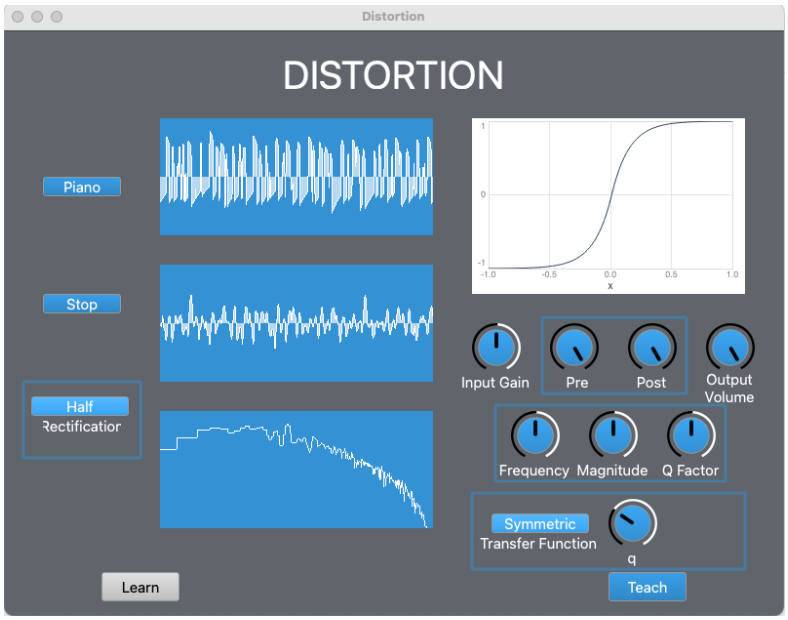

Figure 2. Teach Mode GUI

User Controls:

- Audio Source
  - Sinewave
  - Wav File
- Input Gain
- Pre-Distortion Equaliser
  - Frequency
  - Q
  - Gain
- Pre-Distortion LPF
  - Cutoff Frequency
- Rectifier:
  - Bypass
  - Half Wave Rectification
  - Full Wave Rectification
- Transfer Function Distortion:
  - Bypass
  - Symmetric Transfer Function
    - q = clipping amount
  - Asymmetric Transform Function
    - q = scale range of linear behaviour
    - d = smoothness of transition to clipping
- Post-Distortion Equaliser
  - Frequency
  - Q
  - Gain
- Post-Distortion LPF
  - Frequency
- Output Volume

Signal Path

Figure 3 shows a block diagram of the application's signal path.

The audio source may be either a sine wave oscillator or a WAV file.

The signal is then processed by a parametric equaliser block. With this equaliser the user can boost or cut a band of frequencies from the signal. The signal is later processed by another equalizer that has the opposite behaviour of the first equaliser. For example, if the first equaliser boots frequencies around a centre frequency of 1000Hz by 6 dB, the second equaliser cuts frequencies around the same centre frequency by -6 dB. In a fully linear system this boosting and cutting would result in no change in the overall signal. However, since the Transfer Function implements some non-linear behaviour the boosting and cutting will not fully cancel each other out.

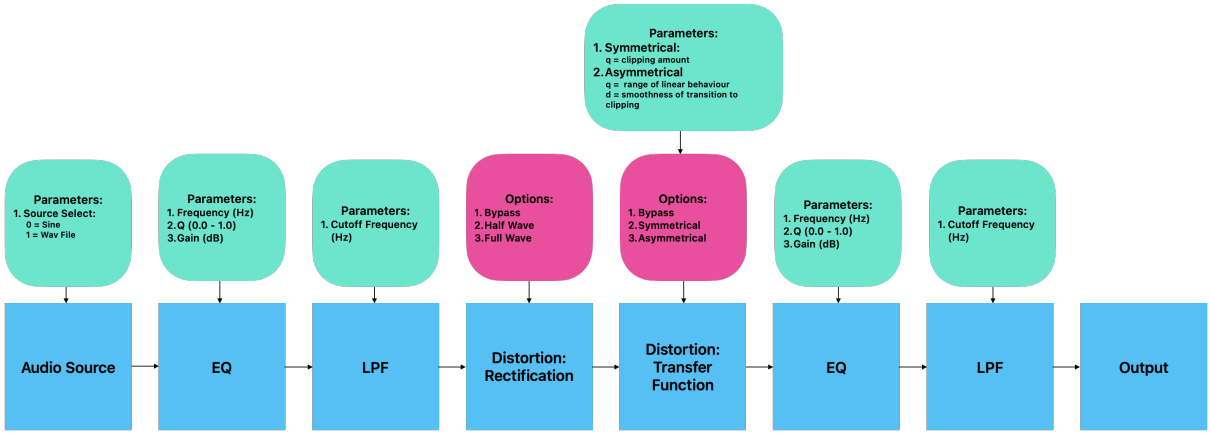

Figure 3. Signal Chain Block Diagram

Following the Pre-Distortion Equaliser, the signal is passed through a Low Pass Filter. Low Pass Filtering may be used in distortion effects to limit the amount of aliasing.

The signal then passes through a rectification stage. Here the user may choose 'Bypass' to remove any effect of rectification or they may choose either Half Wave or Full Wave Rectification.

After rectification the audio signal is processed by the Distortion: Transfer Function block. This block allows the user to choose between a symmetric or asymmetric transfer function to distort the signal. This block has been designed to allow the user real time control over the transfer function's characteristics, for example providing the ability to affect the hardness of the function.

In the last processing stages the signal is passed through the second Equaliser and a second Low Pass Filter. Finally, the signal is sent to the audio output on both left and right channels. A limiter is also included at the output for safety.

Research & Development

After some initial research through the learning material supplied and using online resources, we were aware that the main challenge of this homework would be how to implement the distortion effect using a transfer function. We mainly experimented with three different methods.

Array value modification in the Interpreter

The first method was based on processing the audio signal statically offline in the interpreter. An audio file is read into Supercollider using a buffer. The audio data is then converted into an array. Each sample of the data is then stepped through and processed. The processing was implemented by a series of IF statements. In the example provided in the figure below, we can see that the audio data is processed in three different ways depending on its value.

```
25 ~overdrive = {
26     y = Array.newFrom(a);
27
28     a.do({arg item,i;
29         if ( (abs(item) < 0.33333) && (abs(item) > 0), {y = y.put(i,2*item); "1"});
30         if ( (abs(item) < 0.6666) && (abs(item) >= 0.33333 ), {y = y.put(i,1 - (2-3*item)**0.66666);
   "2"});
31         if ( (abs(item) >= 0.6666) &&( abs(item) <= 1 ), {y = y.put(i,1); "3"});
32         }
33 );
34 "done".postln;
35 };
```

Figure 4. Overdrive effect implemented by affecting values in an array.

The array was then converted back into a buffer before being outputted to the systems audio output. This method achieved our basic needs but did not allow the processing to happen in real time.

Waveshaping with Shaper function

The second method we experimented with used Supercollider's Shaper function. This allowed us to process the audio signal in real time. The Shaper implements a form of waveshaping on the signal using a transfer function in a wavetable format. This method gives a lot of control on the shape of the transfer function as the wavetable is built using an envelope. We were able to create unique transfer functions by simply using some randomization methods within the creation of the envelope.

The results from this method were very pleasing to the ear. However, this method eliminated the option of dynamic waveshaping as the transfer function is implemented as a wavetable in a buffer it's shape cannot be changed in real time.

Mathematical Function Approach

The third method we experimented with used two different custom functions to process the audio data in real time (see Figure 5.). These functions were found in the book Algorithms for Sound Music Computing.

The first equation implements a symmetric transfer function where q gives control over the amount of clipping, higher values proving faster saturation. The second equation implements an asymmetrical transfer function, where q scales the range of the linear behaviour and d controls the smoothness of transition to clipping.

$$F(x) = \operatorname{sgn}(x)\left(1 - e^{-q|x|}\right) \quad F(x) = \frac{x - q}{1 - e^{-d(x-q)}} + \frac{q}{1 - e^{dq}}.$$

Figure 5. Left, Function for symmetric distortion, Right, Function for asymmetric distortion.

Using either of these equations to process each sample of audio data as it is streamed from a buffer, we are able to both distort our signal and control the transfer function's shape in real time. Examples of the transfer functions are shown in figure 6.

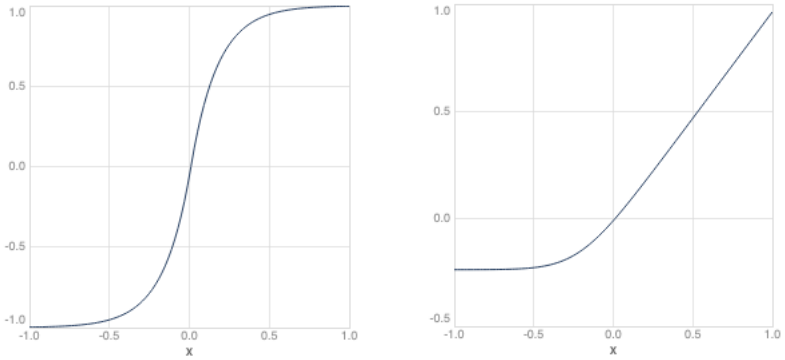

Figure 6. Plotted transfer function for symmetric distortion (left) and for asymmetric distortion (right).

Implementation

This section covers the implementation of each block in our signal chain. As a design choice we coded each of the blocks as individual functions to allow modularity in our programming.

DSP Functionality

~readBuffer reads a WAV file into a buffer from the location *Platform.resourceDir +/+ "sounds/piano.wav"*

~audioSource is used to select the stream of audio data to either be a SinOsc or PlayBuf.

~eqGen implements the SuperCollider function BPeakEQ.

~lpfGen implments the supercollider function LPF.

~rectGen allows the user to choose the kind of rectification for the signal, either None, Half Wave of Full Wave.

~symmetric_distortion implements the function shown on the left of figure 5.

```
416 ~symmetric_distortion = {arg x, q; (x).sign*(1 - (-1*q*(x).abs).exp) };
```

~asymmetric_distortion implements the function shown on the right of figure 5.

```
420 ~asymmetric_distortion = { arg x, q, d; ((x-q) / (1 - (-1*d*(x-q)).exp)) + (q / (1 - ( (d*q).exp) )) };
```

~transF allows the user to choose between no distortion or distortion using either the symmetric or asymmetric transfer function.

The above DSP functionality blocks are then combined in a SynthDef to implement our signal chain.

```
SynthDef(
    "master_",
    { arg freq = 110, eqFreq = 1200, eqQue = 1, eqGain = 12, lpfFreq = 1000, qSym = 8, qAsym = -0.002, dee = 16,
audSelect = 0, postLPFFreq = 1000, inpGain = 1;
        var sourceSig, outputSig, afterPreEQ, afterLPF, afterRect, afterTransF, afterPostEQ, afterLPF2;

        sourceSig = ~audioSource.value(audSelect,freq);
        sourceSig = sourceSig * inpGain;
        afterPreEQ = ~eqGen.value(sourceSig, eqFreq, eqQue, eqGain);
        afterLPF = ~lpfGen.value(afterPreEQ, lpfFreq);
        afterRect = ~rectGen.value(afterLPF, 0);
        afterTransF = ~transF.value(afterRect, qSym, qAsym, dee, 2);
        afterPostEQ = ~eqGen.value(afterTransF, eqFreq, eqQue, -1 * eqGain);
        afterLPF2 = ~lpfGen.value(afterPostEQ, postLPFFreq);
        outputSig = afterLPF2;
        Out.ar([0,1], outputSig)

}).add;

)
```

Figure 7. SynthDef “master_”

GUI

The Graphic Unit Interface is developed in just one window. However, this is divided in three different views, the first one is where the user can understand how to use the application and then choose between one of the two modes, learn or teach, as it is shown in the figure 8. It is important to mention that most of the GUI elements are not visible in the main view by default and are only enabled when one of the two modes is selected.

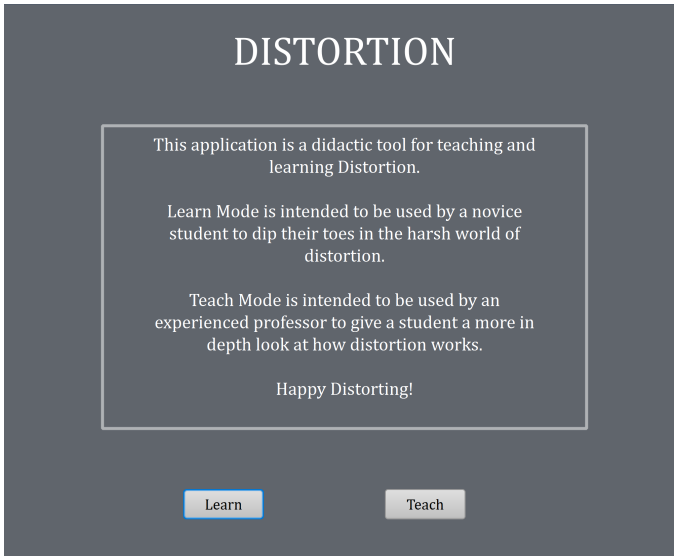

Figure 8. Main view.

The second view is the Learn Mode, characterized with the orange colour. Once the Learn button is active, some GUI objects are visualized, resized, colour changed and relocated as it is shown in the following example. This *visual_(bool)* method allows the user to navigate through the application showing just the objects of interest according to the selected mode. If the Learn or Teach Button are in state 0 (no active) the objects are not visible returning to the main view.

```
//Learn Mode
~learn = Button.new(window, Rect(6*w/20, 16*h/20, 2*w/20, h/20))
.states_([
    ["Learn", Color.black,Color.gray(0.8)], //Satate 0
    ["Learn", Color.white,Color.fromHexString("#EE6C4D")]]) //Satate 1
.action_({
    arg obj;
    if(obj.value == 1,{
        ~teach.valueAction_(0).bounds_(Rect(15.5*w/20, 18.5*h/20, 2*w/20, h/20)); //Teach button set in 0 and actions are triggered
        ~learn.bounds_(Rect(2.5*w/20, 18.5*h/20, 2*w/20, h/20)); //Lear button is moved to the button
        text.visible_(false); //hide main text
        ~explanation.visible = true;
        ~menu.visible = true;

        ~sineSound.states_([["Sine", Color.black,Color.gray(0.8)],
            ["Piano", Color.white,Color.fromHexString("#EE6C4D")]]).visible_(true).action_({ arg butt; ~audioSel = butt.value;
}).valueAction_(0);
        ~playButt.states_([["Play", Color.black,Color.gray(0.8)],
            ["Stop", Color.white,Color.fromHexString("#EE6C4D")]]).value_(0).visible_(true); //Set play button in learn in state 0

        ~ig_knob.bounds_(Rect(4*w/20, 15*h/20, 50, 50)).background_(Color.fromHexString("#EE6C4D")).visible_(true);
        ~lpf_pre_knob.bounds_(Rect(7*w/20, 15*h/20, 50, 50)).background_(Color.fromHexString("#EE6C4D")).visible_(true);
        ~ov_knob.bounds_(Rect(10*w/20, 15*h/20, 50, 50)).background_(Color.fromHexString("#EE6C4D")).visible_(true);

        ~ig_txtknob.bounds_(Rect(3.6*w/20, 16.5*h/20, 2*w/20, h/20)).visible_(true);
        ~lpf_pre_txtknob.bounds_(Rect(6.6*w/20, 16.5*h/20, 2*w/20, h/20)).string_("LPF").visible_(true);
        ~ov_txtknob.bounds_(Rect(9.6*w/20, 16.7*h/20, 2*w/20, h/20)).visible_(true);

        ~scopeView_in.bounds_(Rect(4*w/20, 3*h/20, 7*w/20, 4*h/20)).background_(Color.fromHexString("#EE6C4D")).visible_(true);
        ~scopeView_out.bounds_(Rect(4*w/20, 8*h/20, 7*w/20, 4*h/20)).background_(Color.fromHexString("#EE6C4D")).visible_(true);
        ~trFunction.bounds_(Rect(12*w/20, 4*h/20, 7*w/20, 6*h/20)).visible_(true);
        ~sig.value = ~symmetric_distortion.value(x:Array.series(100,-1, 2/100),q:6); // TF takes the line value pre set
        ~sig.domainSpecs=[-1, 1, \lin, 0, 0, "x"].asSpec; // this set x axis from -1 to 1
        window.drawFunc = { // This function draw the orange square around the knobs
            Pen.strokeColor = Color.fromHexString("#EE6C4D").alpha_(0.5);
            Pen.width = 3;
            Pen.strokeRect(Rect(3.6*w/20, 14.5*h/20, 8*w/20, 3.5*h/20));
        };
        window.refresh;
    }),
```

Figure 9. Code extract showing how the learn view is implemented.

Likewise, this section includes a *Button* object to select the input signal between a sine wave or a piano WAV file. The user can select pre-sets from a *PopUpMenu* between Overdrive, Distortion and Fuzz while the description of these configurations appears in a *StaticText* box, as it is shown in Figure 1. The Input Gain, the LPF and the Output Volume parameters can be modified by changing the values of three knobs respectively. The input and distorted signal are visualized through a *ScopeView* object while the *Plotter* object is used for the Transfer function.

The third view is the Teach Mode, characterized with the blue colour, where the objects of the Learn mode are kept but with some additions. In that way, the *ScopeView* and *Plotter* objects are resized and relocated to add a new frequency domain visual of the output signal with the *FreqScopeView* object.

As the Teach mode gives more options for the user to manipulate the distortion process, there are added two buttons. The first one called Rectification has three states, None, Half and Full. Each time this button is activated, its value selects the corresponding rectification mode through the *action_()* method. This method is important as it is the one in charge to assign the values selected by the user to the corresponding variables and execute the specific functions linked to each object, also to preset the value of some objects like the knobs each time the user press the play button or change the mode.

```
~rectification = Button.new(window, Rect(0.7*w/20, 12.5*h/20, 100, 20)) //Rectification mode button
.states_([["None", Color.black,Color.gray(0.8)],
    ["Half", Color.white,Color.fromHexString("#41B2FD")],
    ["Full", Color.white,Color.fromHexString("#2B73A4")]]).font_(Font("Cambria", 15)).visible_(false)
.action_({ arg butt; ~rectSelect = butt.value;});
```

Figure 10. Three state button for selection of Rectification Mode

The second button, with three states, allows the user to change the Transfer Function of the distortion. For the first state the function is linear/bypass. The second state sets a symmetric distortion function and enables a knob q which controls the hardness of the transfer function. The third state sets an asymmetric transfer function and enables two knobs d and q which control the smoothness of the transition to clipping and the range of the linear behaviour respectively.