POLITECNICO DI MILANO

PROJECT

Ollarizer.

A vocal harmonizer for advanced and beginners musicians.

Autore:
ANDREA EUGENIO LOSI
PAOLO DESANTIS
SILVIO POL

Chapter 1

Brief summary

This project involves the development of a vocal **harmonizer**, a type of pitch shifter that combines the pitch-shifted signal with the original one to create a two or more note harmony. A **pitch shifter** is a sound effects unit that raises or lowers the pitch of an audio signal by a preset interval.

The application has been developed using:

- **SuperCollider** for the audio processing part;
- **Processing** for the implementation of the GUI.
- **OSC Protocol** for the bidirectional communication.

In designing the vocal harmonizer, we aimed to go beyond simply adding two or three voices to the original. The idea was to create an instrument that could be used with a keyboard, giving performers the freedom to manipulate their voice with unlimited, non-fixed interval voices. At the same time, we considered usability for those unfamiliar with keyboards, developing a mode that allows users to select intervals in real time to be sung along with the singer's voice, forming triads and quartets.

Thus, "**ollarizer.**" was created, an instrument designed for both experienced musicians and beginners, allowing a simple toggle switch to shift from one mode to the other.

1.1 GitHub repository

GitHub repository link:

https://github.com/andreaeugeniolosi/Ollarizer.git.

Chapter 2

GUI and functionalities

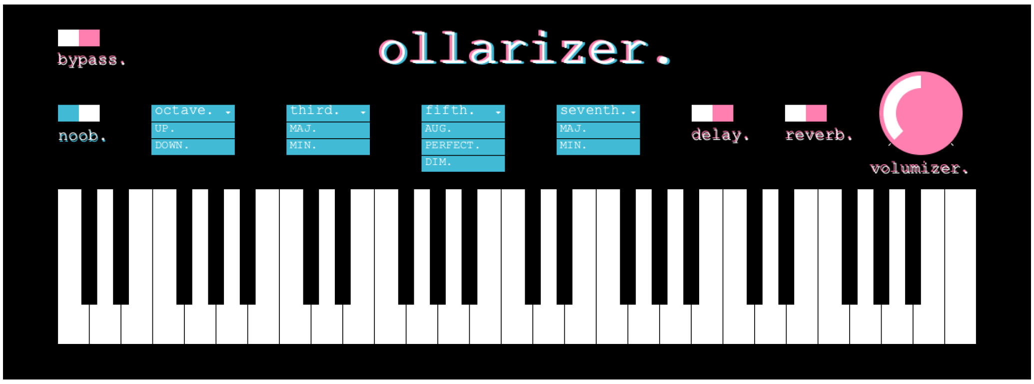

FIGURE 2.1: Graphical user interface

As said in chapter 1, we decided to implement the harmonizer following two parallel approaches that represent two different "version" of the application itself, interchangeable with the `noob` toggle in the GUI.

With the `noob` toggle **activated** the application is meant to be used without a MIDI keyboard, only with a microphone. In this modality you can choose the intervals that you want to be played by your harmonizer (automatically calculated from the note that you are singing). We decided to implement four standard types of intervals, the ones that create the basis of harmony:

- the octave (down or up);
- the third (minor or major);
- the fifth (diminished, perfect or augmented);
- the seventh (minor or major).

The `noob` modality is the default one.

With the `noob` toggle **not activated** the application is meant to be used with a MIDI keyboard and in this modality the application gives the opportunity to sing and play the keyboard in real time. The sound that the keyboard produces is obviously the signal captured by the microphone pitch-shifted for every MIDI note of the keyboard.
It's easy to understand if the `noob` modality is active because the four textboxes of the intervals appear/disappear depending on the value of the toggle.

Let's now analyse the other components of the GUI, the pink ones:

- **"bypass." toggle**: this toggle can be seen as an "ON/OFF" switch. When enabled it deactivates all the other components of the GUI;
- **"delay." toggle**: enables the delay effect;
- **"reverb." toggle**: enables the reverb effect;
- **"volumizer." knob**: sets the volume of the voices when using the vocal harmonizer;
- the **midi keyboard**: when not in "noob" mode, the midi keyboard is a representation of the physical keyboard. The keys change color when pressed.

Chapter 3

Supercollider

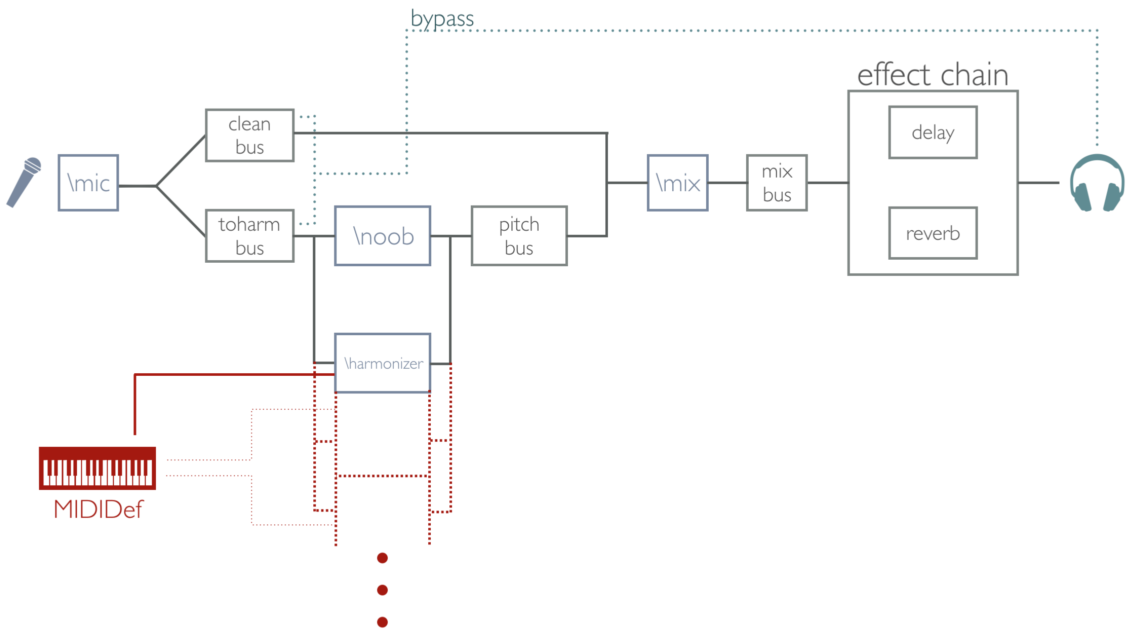

FIGURE 3.1: SC Block diagram

All we need to start is an input audio signal from a microphone, read in supercollider by the `SynthDef \mic` with the methods `SoundIn`; knowing that the "ollarizer." will be mostly used by vocalists, we decided to highpass the incoming signal avoiding some very low frequency noises that could be adversely affect the spectrum of the voice, compromising the sound quality and interfering with pitch estimation.
We wanted that the output should be the harmonized voices but also the clean one of the user, so we allocated the outcomes of the `\mic` in two `Audio.Bus`: one that will not be pitched and one that will be sent to the `SynthDef \harmonizer` where the main function `PitchShift` takes place.
To achieve a dynamic harmonizer we needed a dynamic pitch shifting who recognizes the notes played and changes their frequencies according to the MIDI inputs the client gives; first we need to evaluate the pitch of the input signal and, in order to do that, instead of using the simple class `Pitch`, which works in time domain, we used the class `Tartini`, which performs much better working in frequency domain (we had also insert a compander as a gate, with clean voice as control signal, to interrupt the estimation when the voice turns down of volume and the detection is impossible, due to the fact that the frequency picks are too low to be relevant). Obtained this value, now we have to feed `PitchShift` with the right ratio of shifting for every sung note: $ratio = MIDI\_note\_frequency/estimated\_frequency$.
The next step was to give to `SynthDef \harmonizer` a MIDI note: `MIDINoteOn` and `MIDINoteOff` came to our aid. We created an array that has the number of elements equal to the number

of the MIDI notes's totality and inside `MIDINoteOn` we fed it with the `SynthDef \harmonizer` output values in the correspondent index to the MIDI note pressed; contrariwise in `MIDINoteOff` we free the memory of the index releasing the same synth.
As mentioned in chapter 1, the "noob" harmonizer, instead of processing MIDI inputs, simply pitch-shift the input voice track with absolute frequency ratio.
Then the outcome is stored in the `pitchBus` and mixed with the previous clean signal in `SynthDef \mix`, where the class `Mix` ensures that the signal is correctly normalized and doesn't saturate, and allows us to control the intensity of the voices from the "volumizer." knob.
The last part of the processing is an effects chain composed by a short delay and a dry reverb to make the sound more pleasurable.

Our Supercollider Server communicates with the Processing GUI through OSC protocol, therefore translating the user interaction in different parameter settings of our Synth; we instantiated an OSC `listener` that once receiving the OSC messages it parses and calls the matched `.set()` functions. We analyze better this topic in chapter 4.

Chapter 4

Open Sound Control Communication

To ensure effective bidirectional communication between the user interface and the SuperCollider server, we adopted the Open Sound Control (OSC) protocol. This approach allows not only sending commands from Processing to SuperCollider but also making the GUI responsive to OSC messages sent by SuperCollider. The OSC communication management was implemented using a structured approach that facilitates handling harmonizer parameters and real-time visualization of MIDI inputs.

4.1 Processing

Processing allowed us to easily manage the Open Sound Control with its native libraries `oscP5` and `netP5`. We implemented all the communication managing within the `oscHandler.PDE` file that, firstly initializes the `NetAddress` (listening and sending IPs and Socket Ports) establishing the connection with Supercollider, then, on the user interaction inputs, the `sendOSCMessagge(String addressPattern, int value)` function builds properly the OSC messages and sends them to SC through the socket port.
Moreover, we wanted Processing to be responsive to the MIDI `noteOn/notOff` messages collected and sent by SC to our Processing app. The function `oscEvent(OscMessage theOscMessage)` parses the incoming OSC messages and, through the `midiHandler` methods, our application is able to process **in real time** the MIDI messages and makes them visible through the color changing of the GUI keyboard keys.

4.2 Supercollider

SC implements by itself the OSC Protocol handling functions. It was easy to initialize the connection with processing and through the method `thisProcess.addOSCRecvFunc( listener);` we istantiated a listener that receives and parses the OSC messages and manages the Synth parameter settings.