ASSESSMENT 2 - AUDIO PROGRAMMING 2

ABSTRACT

The report illustrates how has been built a VST3 audio plugin, using the JUCE Framework (C++) which implements a stereo 'ping-pong' delay effect. The Plugin offers to the user the possibility to control its parameters by an User Interface, which is able to change their values while the program is running and to save its state when the DAW is open or closed.

Once that the design of the Plugin has been done and the VST3 file has been generated, it has been open on the Fruit Loop Studio D.A.W., where it has been performed and it has described any details of any user controls and how they impact on the resulting audio process.

Table of contents

1 INTRODUCTION

1.1 Overview

The ping pong delay effect is implemented using the Juce Framework, as a VST3 Plugin for stereo Input/Output. A stereo ping-pong delay seeks to create an interesting effect across the stereo soundstage by repeating and overlapping a section (or sections) of input stereo audio in such a way that the delayed (processed) left channel audio input appears on the right output channel and the delayed (processed) right channel audio input appears on the left output channel.

As a default Juce setting, we have two processors, working at the same time, one that handles the ping pong effect and all the signal processing of the audio data, and the other one which handles the User Interface, where all the parameters can be controlled from the user.

This is a system with memory, so we need to implement a “delay line” in our process block, because the output does not depend only of its own input, but also on the input signal of the past.

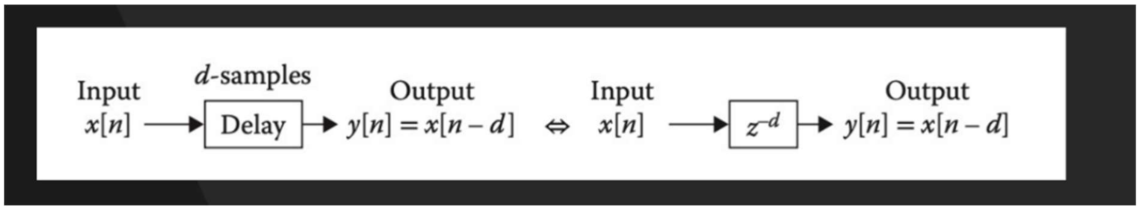
1.2 Background theory

1.2.1 SYSTEMS WITH MEMORY

A memory-less system only depends on the input to the system at the same time. These systems do not require the memory of previous time samples to process a signal. However, many signal processing systems used in audio where the output at any time can depend on the input signal at a previous time. These systems require memory to store the input samples for later use. These are *systems with memory.* Systems with memory have the ability to temporarily store a signal and delay it over time.

1.2.2 DELAY BLOCK SYSTEMS

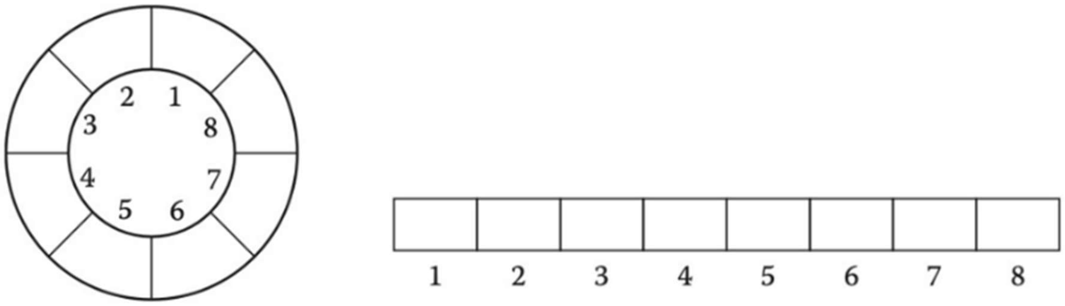
Systems with memory have the ability to temporarily store a signal and delay it over time. A delay processing block receives a sample of the input signal at sample number n and stores this sample in memory for some number of samples, d. At sample number $n + d$, the output of a delay processing block is the input sample, $x[n]$. Therefore, when input sample $x[n]$ is received by a delay processing block, the output of the block at the same time is the sample $x[n - d]$, or the sample stored in memory for d samples
In order to simplify the display of a block diagram, the notation z^{-d} is used to represent a delay block of d samples. This convention comes from the z-plane. For a system to "remember" past audio samples you need to be able store them in a computers memory.
A collection of audio samples stored for the purpose of reading them back at a later date is called a *delay buffer* or *delay line.*

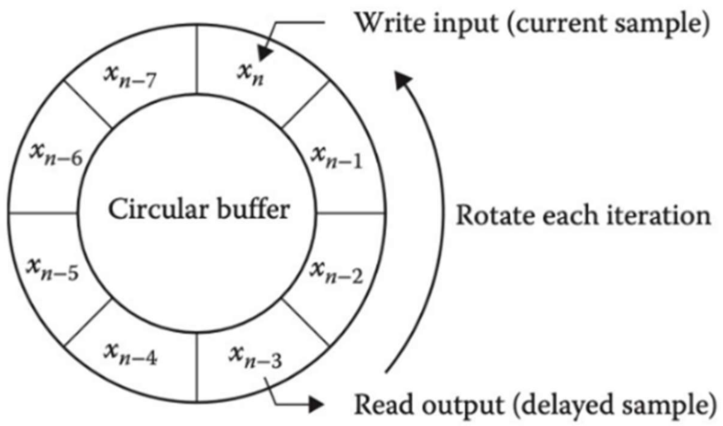


1.2.3 CIRCULAR BUFFER

In a circular buffer, the value of each element is stored one time in its location and does not shift during an iteration.

Instead, the relative location of the start of the buffer is changed during an iteration. Therefore, it is only necessary to change one value (the buffer index) during an iteration instead of every element of the buffer. The index of the delayed sample is determined relative to the current buffer index.





1.2.4 VALUE TREE STATE PARAMETER CLASS

AudioProcessorValueTreeState is a class which contains a ValueTreeObject which can store all of your audio parameters in a single structure. This makes the implementation and connection of audio parameters and graphics objects in the editor much easier and also makes saving and loading plug-in states easier. The AudioProcessorValueTreeState also has automatic undo/redo functionality. This is all done in a thread safe way (meaning there is no problems between the processor running at a higher priority thread than the editor on a lower priority).

1.2.5 SMART POINTERS, UNIQUE POINTERS

Smart pointers are used to make sure that an object is deleted if it is no longer used (referenced).

std::unique_ptr is a smart pointer that owns and manages another object through a pointer and disposes of that object when the unique_ptr goes out of scope.

The object is disposed of, using the associated deleter when either of the following happens:

the managing unique_ptr object is destroyed

the managing unique_ptr object is assigned another pointer via operator= or reset().
The object is disposed of, using a potentially user-supplied deleter by calling get_deleter()(ptr). The default deleter uses the delete operator, which destroys the object and deallocates the memory.

A unique_ptr may alternatively own no object, in which case it is called *empty*.

There are two versions of std::unique_ptr:

1. Manages a single object (e.g. allocated with new)
2. Manages a dynamically-allocated array of objects (e.g. allocated with new[])

The class satisfies the requirements of "MoveConstructible", and "MoveAssignable", but of neither "CopyConstructible" nor "CopyAssignable".

1.2.6 THREADS AND DATA RACE CONDITIONS->std::Atomic

A thread of execution is a flow of control within a program that begins with the invocation of a top-level function by std::thread::thread, std::async, or other means.

Any thread can potentially access any object in the program (objects with automatic and thread-local storage duration may still be accessed by another thread through a pointer or by reference).

Different threads of execution are always allowed to access (read and modify) different memory locations concurrently, with no interference and no synchronization requirements.

When an evaluation of an expression writes to a memory location and another evaluation reads or modifies the same memory location, the expressions are said to conflict. A program that has two conflicting evaluations has a data race unless both evaluations execute on the same thread or in the same signal handler, or both conflicting evaluations are atomic operations (see std::atomic).

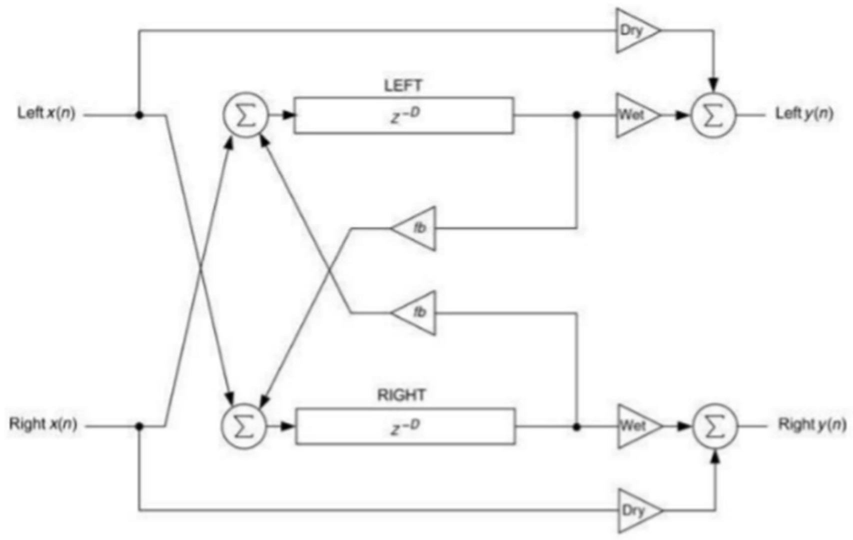
Each instantiation and full specialization of the std::atomic template defines an atomic type. If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined (see memory model for details on data races).

1.2.7 XML FILE

An XML file is an extensible markup language file, and it is used to structure data for storage and transport. In an XML file, there are both tags and text. The tags provide the structure to the data. The text in the file that you wish to store is surrounded by these tags, which adhere to specific syntax guidelines. At its core, an XML file is a standard text file that utilizes customized tags, to describe the structure of the document and how it should be stored and transported.

All kinds of applications can benefit from XML as it offers a streamlined method of accessing information. This straightforward process allows both applications and devices to utilize, store, transfer and display data. For example, in the workplace data architects and programmers use XML daily.

METHODOLOGY



2.1 Implementation of the ping pong effect

The actual implementation of our effect is been done in our call back function, the process block. We created some global variables in our Processor class:

```
 AudioBuffer<float> delayBuffer;
AudioBuffer<float> wetBuffer;
int writePosition { 0 };
int localSampleRate { 44100 };

int maxDelay { 2 }.
```

We created two constant variables “buffer length” and “delay buffer length” and put them equal to the number of samples of our block.

Inside our loop function, where we are iterating through our two channels, we created three constant float pointers, corresponding with the buffer data of each buffer.

The process has been divided in three parts, using a function for each of it:

- Filler buffer function;

- Get From Delay Function;
- Feedback Delay Function.

In the first function It has been actually copied the data from our main buffer to the delay buffer.

In the second It has been actually created the delayed signal and copied it in a new buffer, called wet buffer, and then added to our main buffer at the end.

In the third function actually creates the eco effect that we can hear from the speakers, adding the "wet data" to the delay buffer.

2.1.1 AUDIO BUFFER MEMBER FUNCTIONS USED

- **Get Read Pointer (channel):**

Returns a pointer to an array of read-only samples in one of the buffer's channels.

- **addFromWithRamp(….):**

Adds samples from an array of floats, applying a gain ramp to them.

Parameters**:**

destChannel the channel within this buffer to add the samples to

destStartSample the start sample within this buffer's channel

source the source data to use

numSamples the number of samples to process

startGain the gain to apply to the first sample (this is multiplied with the source samples before they are added to this buffer)

endGain the gain to apply to the final sample. The gain is linearly interpolated between the first and last samples.

- **copyFromWithRamp(….):**

Copies samples from an array of floats into one of the channels, applying a gain ramp.

The hasBeenCleared method will return false after this call if samples have been copied.

Parameters:

destChannel	the channel within this buffer to copy the samples to
destStartSample	the start sample within this buffer's channel
source	the source buffer to read from
numSamples	the number of samples to process
startGain	the gain to apply to the first sample (this is multiplied with the source samples before they are copied to this buffer)
endGain	the gain to apply to the final sample. The gain is linearly interpolated between the first and last samples.

- **CopyFrom(….)**:

  Copies samples from an array of floats into one of the channels, applying a gain to it.

  Parameters

destChannel	the channel within this buffer to copy the samples to
destStartSample	the start sample within this buffer's channel
source	the source buffer to read from
numSamples	the number of samples to process
gain	the gain to apply

- **AddFrom(….)**:

  Adds samples from another buffer to this one.

  Parameters

  destChannel the channel within this buffer to add the samples to destStartSample the start sample within this buffer's channel

source the source buffer to add from

sourceChannel the channel within the source buffer to read from

sourceStartSample the offset within the source buffer's channel to start reading samples from

numSamples the number of samples to process

gainToApplyToSource an optional gain to apply to the source samples before they are added to this buffer's samples

2.2 User Interface

2.2.1 CREATION OF THE VALUE TREE STATE PARAMETERS

First of all, It has been created three variables in the Processor class, “feedback, bpm and level”, the actual parameters which the user will control and a AudioProcessorValueTreeState object called “parameters”, where all the parameters settings are stored. In the constructor it has created and added parameters to our processor and we put our three variables equals to “get raw parameter value” function”, which is how they are connected to the Valuee Tree state processor which can read to find their current values.

- **getRawParameterValue**()

std::atomic<float>*
AudioProcessorValueTreeState::getRawParameterValue (
StringRef parameterID)

Returns a pointer to a floating point representation of a particular parameter which a realtime process can read to find out its current value.

- **createAndAddParameter**()

RangedAudioParameter* AudioProcessorValueTreeState::createAndAddParameter (
    std::unique_ptr< RangedAudioParameter > parameter)

This function adds a parameter to the attached AudioProcessor and that parameter will be managed by this AudioProcessorValueTreeState object.

- **getParameterRange()**

NormalisableRange<float> AudioProcessorValueTreeState::getParameterRange (
    StringRef parameterID)

Returns the range that was set when the given parameter was created.

2.2.2 AUDIO PROCESSOR EDITOR

In the audio processor editor class it has been created three scoped pointer of the value tree state object "Slider Attachment" and also three slider to control the parameters. In the constructor, sliders are connected to parameters; here there are also the functions needed to set all the visual information of the sliders such as size, text, label etc. It has used scoped pointer here, instead of Unique pointers, so it has inserted the three slider attach values in the distructor, as they are not able to auto delete their self.

- **SliderAttachment()**

An object of this class maintains a connection between a Slider and a parameter in an AudioProcessorValueTreeState. During the lifetime of this SliderAttachment object, it keeps the two things in sync, making it easy to connect a slider to a parameter. When this object is deleted, the connection is broken.

2.2.3 PARAMETER STATE INFORMATION:XML DOCUMENT

In order to load and save the state of parameters when the DAW is open and closed, it has been implemented a XML document, where we can store the parameter s information. This process, has been done inside the two functions "get state information" and "set state information", where we have created the XML document and we have actually restore our parameters state.

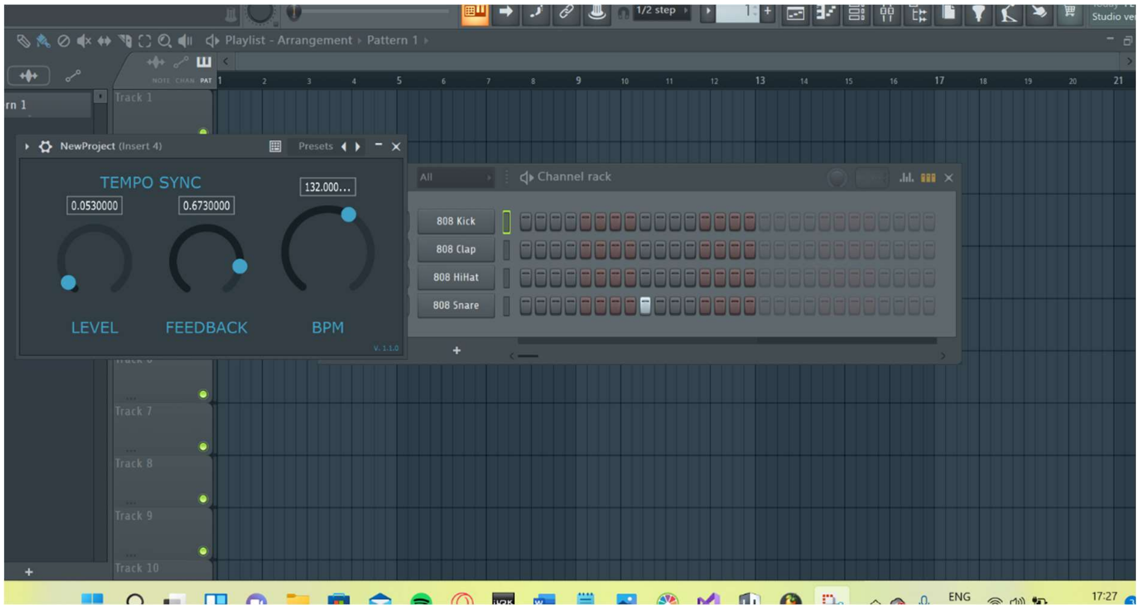
RESULTS AND CONCLUSION

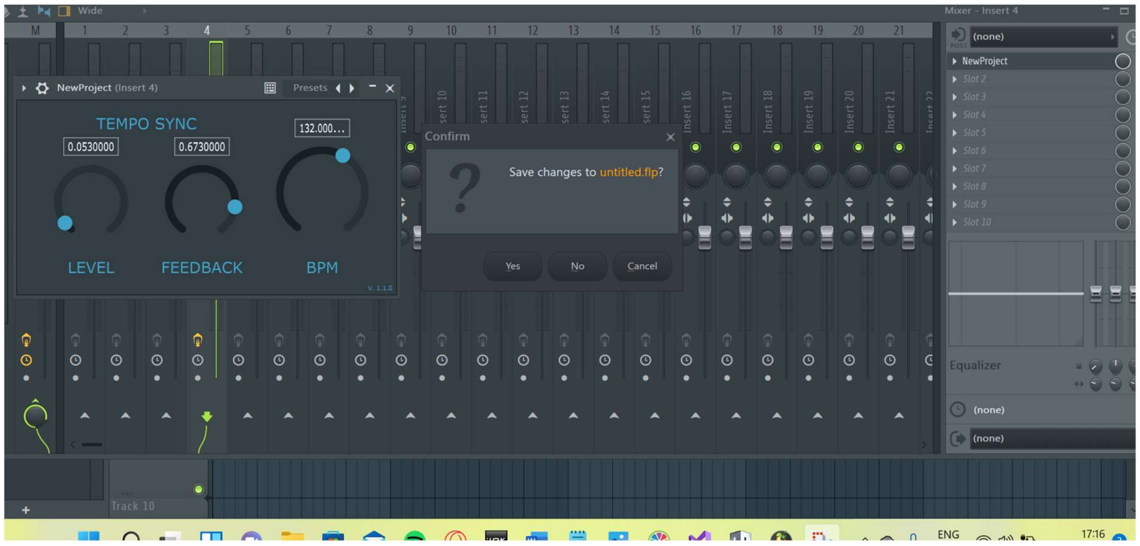
3.1 Testing the Audio Processing performance

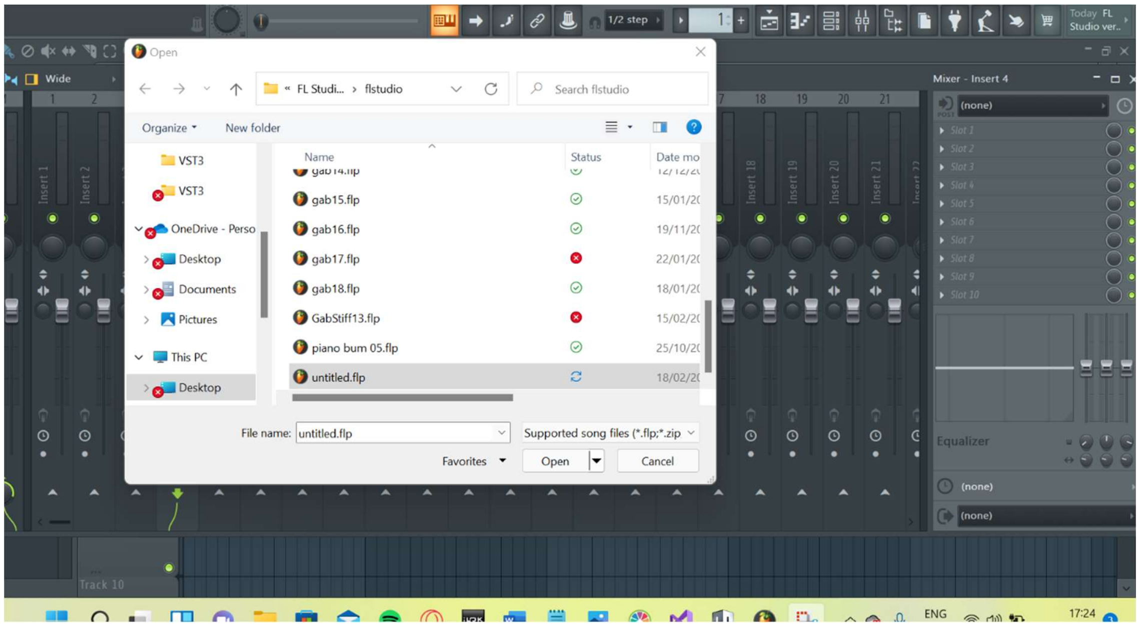
After have completed all the implementation and have built and launched successfully the project, it has been loaded on the DAW FL Studio. It has been played a drum pattern and the Plugin is been loaded on the snare channel of the mixer.

The Plugin seems to perform a ping pong delay effect, because we have typed an equation who make the program send double delay time in one channel compare to the other. All the three knobs work properly. the "level knob", controls the gain of the signal transferred to our delay buffer from the main one; in fact we heard a softer delay decreasing it and, if It is put equal to zero it is played only the dry signal. The "BPM knob" make the delay time rise the more we decries it, and vice versa, the delay time reach the maximum of 2 seconds when the BPM is equal to 30 (minimum BPM value). The feedback knob, controls how much amount of eco effect we hear from the speaker; putting it equal to zero, it is played only the dry signal with the delay line, but without repeating it over the time.

3.2 Testing the saved/loaded state of the Plugin

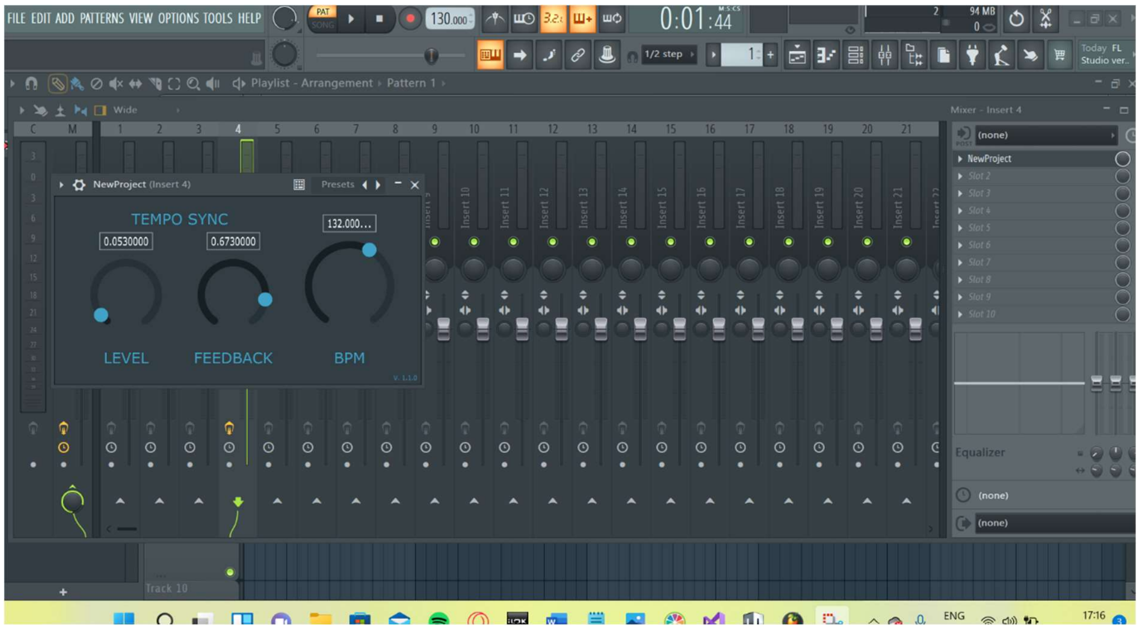






Name	Status	Date mo
gab14.flp	✓	12/12/20
gab15.flp	✓	15/01/20
gab16.flp	✓	19/11/20
gab17.flp	✗	22/01/20
gab18.flp	✓	18/01/20
GabStiff13.flp	✗	15/02/20
piano bum 05.flp	✓	25/10/20
untitled.flp	⟳	18/02/20

File name: untitled.flp
Supported song files (*.flp;*.zip


FILE EDIT ADD PATTERNS VIEW OPTIONS TOOLS HELP
Playlist - Arrangement › Pattern 1
Mixer - Insert 4
NewProject (Insert 4)
Presets
TEMPO SYNC
Equalizer
Track 10

LEVEL 0.0530000
FEEDBACK 0.6730000
BPM 132.000...

3.3 Summary

The project is a basic version of VST3 Stereo Ping Pong Delay Plugin that performs his tasks correctly, but it is still in a beta version; both the Digital Signal Processing and the User Interface aspects might be improved. An example could be when the knobs are moved quickly: we notice some discontinuities, such as crackles and clicks that could be removed using a Smoother Parameter filter. We could also implement Low Pass filters, distortions to the feedback signals or to incorporate further delay lines to generate more temporally dense repetitions of input.