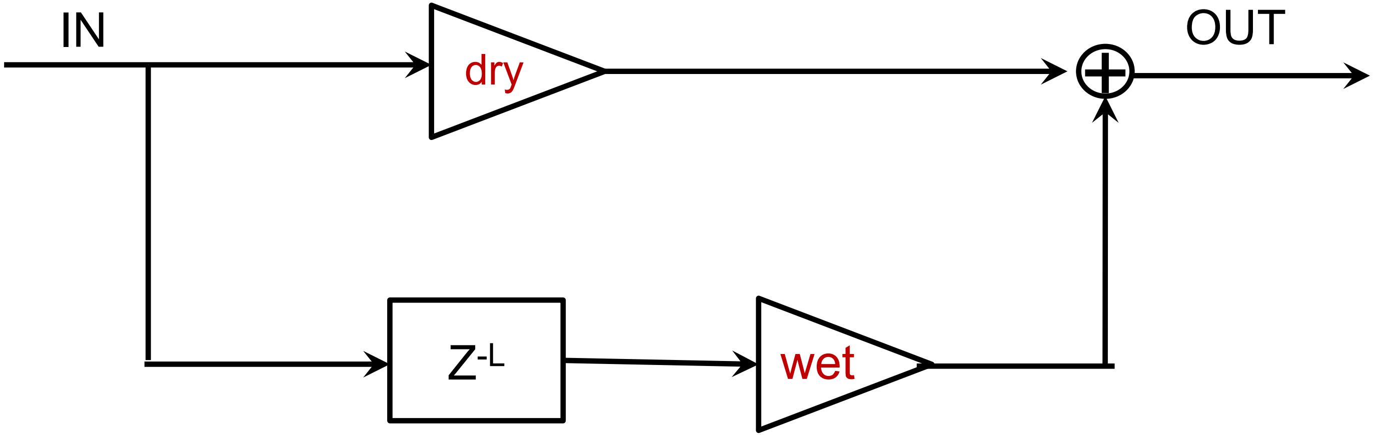
POLITECNICO
MILANO 1863

ISPL
Image and Sound Processing Lab

JUCE exercises

Ex 2: Delay Line

In this second example we are going to implement a **simple delay**.



The effect must allow the user to control of the *wet* and *dry* coefficients, and of the delay *L.* This is accomplished through sliders.

Delay Line: Audio Buffer

- The audio input/output is managed by buffers.
- The size of the audio buffer is not set within the plugin, but it is inherited from the DAW. As a consequence in the plugin we need to retrieve its current dimension.
- The audio buffer is a matrix whose size is

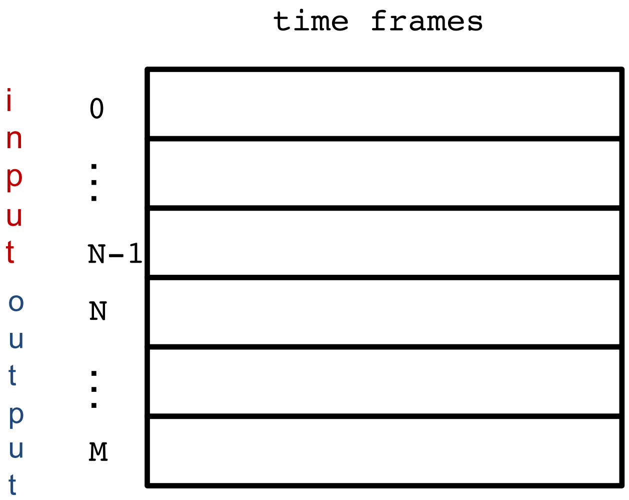
  (totalInputChannels + totalOutputChannels) x timeFramesInEachBlock

- The first and second indexes of AudioBuffer point to the channel and sample, respectively.
  - The *k*th sample in the *i*th input channel is at

    buffer[i-1][k-1].

  - The *k*th sample in the *i*th output channel is at

    buffer[i-1+totalInputChannels][k-1].



Delay Line: Audio Buffer

- The user can also declare and use buffers which are not linked to any input / output channels. This is useful in our case for implementing the delay line (bottom branch of the block diagram).
- This is implemented by the `AudioSampleBuffer` class
- some of the methods available for this class are:
    - setting the buffer: `setSize(numChannels, blockSize); clear();`
    - reading a sample in the buffer `getSample(channel, sampleIndex);`
    - writing a sample in the buffer: `setSample(channel, sampleIndex);`

Delay Line

As we did for the first exercise, we need to perform 3 main steps:

- Create GUI control
- Pass control information to the Processor class
- In the Processor class process the audio input to implement the delay line

Let's create a new **audio plug-in** in Projucer, and call it `DelayLine`.

Delay Line Editor

1) Create the GUI control:

1 • In the declaration, create a `Slider` object as a private member of the class `DelayLineAudioProcessorEditor` (let's call it `Editor` from now on);

```
juce::Slider wetSlider;
juce::Label wetLabel;
juce::Slider drySlider;
juce::Label dryLabel;
juce::Slider timeSlider;
juce::Label timeLabel;
```

Delay Line Editor

2 • Make the class Listener as base class

```
class DelayLineAudioProcessorEditor
:public juce::AudioProcessorEditor,
private juce::Slider::Listener
```

3 • Add the declaration of the function to be called when one of the Slider change

```
void sliderValueChanged(juce::Slider* slider) override;
```

Delay Line Editor

4 • In the constructor of the Editor, set all the properties for the Sliders and the Labels + add listener

```
wetSlider.setRange (0.0, 1.0);
wetSlider.setTextBoxStyle (juce::Slider::TextBoxRight, false, 100, 20);
wetSlider.addListener(this);
wetLabel.setText ("Wet Level", juce::dontSendNotification);
addAndMakeVisible (wetSlider);
addAndMakeVisible (wetLabel);
drySlider.setRange (0.0, 1.0);
drySlider.setTextBoxStyle (juce::Slider::TextBoxRight, false, 100, 20);
drySlider.addListener(this);
dryLabel.setText ("Dry Level", juce::dontSendNotification);
addAndMakeVisible (drySlider);
addAndMakeVisible (dryLabel);
timeSlider.setRange (500, 50000, 100);
timeSlider.setTextBoxStyle (juce::Slider::TextBoxRight, false, 100, 20);
timeSlider.addListener(this);
timeLabel.setText ("Time", juce::dontSendNotification);
addAndMakeVisible (timeSlider);
addAndMakeVisible (timeLabel);
setSize (400, 300);
```

Delay Line Editor

5 • In the function resized of the Editor, set the position of both Labels and Sliders

```
wetLabel.setBounds (10, 10, 90, 20);
wetSlider.setBounds (100, 10, getWidth() - 110, 20);

dryLabel.setBounds (10, 50, 90, 20);
drySlider.setBounds (100, 50, getWidth() - 110, 20);

timeLabel.setBounds (10, 90, 90, 20);
timeSlider.setBounds (100, 90, getWidth() - 110, 20);
```

6 • Define the function to be called every time a Slider change value

```
void DelayLineAudioProcessorEditor::sliderValueChanged(juce::Slider *slider)
{
    if (slider == &wetSlider)
        audioProcessor.set_wet(wetSlider.getValue());
    else if (slider == &drySlider)
        audioProcessor.set_dry(drySlider.getValue());
    else if (slider == &timeSlider)
        audioProcessor.set_ds(timeSlider.getValue());
}
```

Delay Line Processor

2) Pass the information to the Processor class

7 - Add to the `Processor` definition as a private member the variables `wet`, `dry` and `ds` (needed to echange information) and an `AudioSampleBuffer dbuf, dr` read and `dw` write indexes

```
juce::AudioSampleBuffer dbuf;
int dw ;
int dr;

float wet;
float dry;
int ds;
```

Delay Line Processor

8 • In the public section of `Processor` declare the function to be called to set the values of `wet`, `dry` and `ds`

```
void set_wet(float val);
void set_dry(float val);
void set_ds(int val);
```

9 • In the implementation of the `Processor` actually define these functions

```
void DelayLineAudioProcessor::set_wet(float val)
{
    wet = val;
}
void DelayLineAudioProcessor::set_dry(float val)
{
    dry = val;
}
void DelayLineAudioProcessor::set_ds(int val)
{
    ds = val;
}
```

Delay Line

10 • in the `Processor` method `prepareToPlay()` set the `AudioSampleBuffer` properties and set the initial values for delay value, read and write indexes;

```
dbuf.setSize(getTotalNumOutputChannels(), 100000);
    dbuf.clear();

    dw = 0;
    dr = 1;
    ds = 50000;
 }
```

Delay Line

11 • In the `Processor` method `processBlock()`:
  - clear the buffers (already in the default code) and update the values of wet, dry and delay number of samples;
  - get two pointers for writing in the output channels and one for reading from the input channels;
  - sample by sample of the audio buffer (`for` cycle);
    - store in `dbuf` the current sample, in position `dw`;
    - compute the mix of dry and wet; for the dry component, use the read pointer for the input channel (upper line in the scheme); for the wet component, use the `dbuf` buffer and index `dr` (repeat for each output channel)
    - update the indexes `dr` and `dw` using the modulo operation wrt delay value `ds`

Delay Line

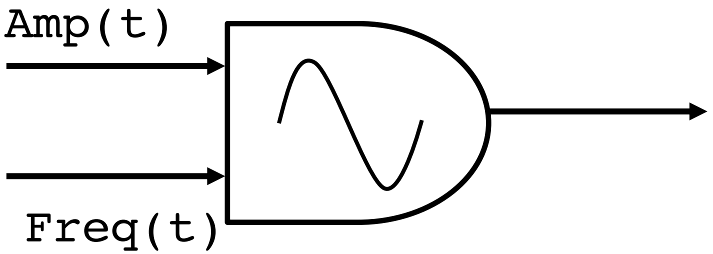
```
int numSamples = buffer.getNumSamples();
    float wet_now = wet;
    float dry_now = dry;
    int ds_now = ds;
    float* channelOutDataL = buffer.getWritePointer(0);
    float* channelOutDataR = buffer.getWritePointer(1);
    const float* channelInData = buffer.getReadPointer(0);
    for (int i = 0; i < numSamples; ++i) {
        // setSample(int destChannel, int destSample, Type newValue)
        dbuf.setSample(0, dw, channelInData[i]);
        dbuf.setSample(1, dw, channelInData[i]);
        channelOutDataL[i] = dry_now * channelInData[i] + wet_now * dbuf.getSample(0, dr);
        channelOutDataR[i] = dry_now * channelInData[i] + wet_now * dbuf.getSample(1, dr);
        dw = (dw + 1 ) % ds_now ;
        dr = (dr + 1 ) % ds_now ;
    }
```

Delay Line: test with Audio Plugin Host

Ex 3: a simple Oscillator

In this example we are going to write a simple oscillator where frequency and amplitude are controlled by two rotary sliders.

$$y(t) = a(t)\sin\big(\phi(t)\big)$$
$$\phi(t) = 2\pi f_0 t$$



Oscillator

We are going to implement together both the GUI control and the audio processing algorithm.

Let’s create a project using Plugin template called `Oscillator`;

Oscillator GUI

Exercise:

Implement the GUI part

- Add to the GUI two Rotary sliders for frequency and amplitude and the two corresponding Labels
- Frequency should vary between 50 and 500, amplitude between 0 and 1
- Implement the mechanism of control exchange between the Editor and the Processor; remember the Editor should always take care of updating the Processor.

Oscillator Processor

1 • In the `Processor` declaration add a couple of constants we are going to use during the implementation of the oscillator

```
#define SAMPLE_RATE  (44100)
#ifndef M_PI
#define M_PI  (3.14159265)
#endif
```

2 • As private member, add a variable that will be used for sinusoid computation

```
float phase;
```

Oscillator Processor

3 • in the method `prepareToPlay` of the `Processor` class we initialize the variables for the oscillator (freq and amplitude are the ones used for control exchange)

```
freq = 440.0;
amplitude = 0.0;
phase = 0.0;
```

Oscillator Processor

4 • now we are ready for modify the `processBlock` method of the `Processor` class

- prepare two writing pointers for the left and right channels

```
float* channelDataL = buffer.getWritePointer(0);
float* channelDataR = buffer.getWritePointer(1);
```

- store the current values of frequency and amplitude and retrieve the number of samples of the audio buffer

```
float amplitude_now = amplitude;
float freq_now = freq;
int numSamples = buffer.getNumSamples();
```

Oscillator Processor

- sample by sample (`for` cycle) we update the values for the argument of the sinusoid, using the `freq` and `amplitude` values provided by the `Sliders`.

```
for (int i = 0; i < numSamples; ++i) {
    channelDataL[i] = amplitude_now * (float) sin ((double) phase);
    channelDataR[i] = channelDataL[i];

    phase +=  (float) ( M_PI * 2. *( ((double) freq_now  / (double)
SAMPLE_RATE)));

    if( phase > M_PI * 2. ) phase -= M_PI * 2.;
}
```

Sin Argument Computation

The argument of the sinusoid is iteratively incremented:

$$y(t_i) = a(t_i)\sin(\phi(t_i))$$

$$\phi(t_i) = 2\pi f_0 t_i = 2\pi f_0 (t_{i-1} + \Delta t) = 2\pi f_0 t_{i-1} + 2\pi f_0 \, \Delta t$$

Audio Plugin Host configuration